



University of **HUDDERSFIELD**

University of Huddersfield Repository

Bonner, Stephen Arthur Robert

Using Hadoop to implement a semantic method for assessing the quality of medical data

Original Citation

Bonner, Stephen Arthur Robert (2014) Using Hadoop to implement a semantic method for assessing the quality of medical data. Masters thesis, University of Huddersfield.

This version is available at <http://eprints.hud.ac.uk/id/eprint/23305/>

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

<http://eprints.hud.ac.uk/>

UNIVERSITY OF HUDDERSFIELD

Using Hadoop To Implement A Semantic Method For Assessing The Quality Of Medical Data

Author:

Stephen BONNER

Supervisor:

Prof. Grigoris ANTONIOU

*A thesis submitted to the University of Huddersfield
in partial fulfilment of the requirements for the degree of Masters By Research*

High Performance Computing
School Of Computing and Engineering

July 2014



Copyright Statement

- i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the "Copyright") and s/he has given The University of Huddersfield the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.
- ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the University Library. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.
- iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the "Intellectual Property Rights") and any reproductions of copyright works, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

"Your time is limited, so don't waste it living someone else's life. Don't be trapped by dogma - which is living with the results of other people's thinking. Don't let the noise of others' opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition."

Steve Jobs 2005

"The secret of the mountain is that the mountains simply exist, as I do myself: the mountains exist simply, which I do not. The mountains have no "meaning," they are meaning; the mountains are. I understand all this, not in my mind but in my heart, knowing how meaningless it is to try to capture what cannot be expressed, knowing that mere words will remain when I read it all again, another day."

Peter Matthiessen 1978

"If you don't know, the thing to do is not to get scared, but to learn."

Ayn Rand 1957

Abstract

Recent technological advances in modern healthcare have lead to a vast wealth of patient data being collected. This data is not only utilised for diagnosis but also has the potential to be used for medical research. However, there are often many errors in datasets used for medical research, with one study finding error rates ranging from 2.3% to 26.9% in a selection of medical research databases.

Previous methods of automatically assessing data quality have often relied on threshold rules. These rules can sometimes miss errors requiring complex domain knowledge to correctly identify. To combat this, a semantic framework has been developed to assess the quality of medical data expressed in the form of linked open data. Early work in this direction revealed that existing triplestores are unable to cope with the large amounts of medical data.

In this thesis, a system for storing and querying medical RDF data using Hadoop is developed. This approach enables the creation of an inherently parallel framework that will scale the workload across a cluster. Unlike existing solutions, this framework uses highly optimised joining strategies to enable the completion of eight separate SPARQL queries, comprising over eighty distinct joins, in only two Map/Reduce iterations. Results are presented comparing both naïve and optimised versions of the solution against Jena TDB, demonstrating the superior performance of the Hadoop system and its viability for assessing the quality of medical data.

Acknowledgements

I would like to thank the many people who have supported and helped me with this thesis. I would firstly like to thank my parents, Dr Louise Bonner and Mr Neil Bonner, for all their incredible and unwavering support during my academic career. I would also like to thank my supervisory team of Professor Grigoris Antoniou and Dr Violeta Holmes. I would also like to thank Dr Laura Moss and Dr David Corsar for their original effort in this work and for allowing me to expand upon it.

I would like to thank all of my family and friends, of which there are too many to mention in full here, but I would like to mention Holly Lewis, Joseph Luke, Olive Chittock, Liam lane, Professor Fiona Tweed, Dave Guldin, Barbara Hughes, Dr John Ambrose and Neil MacGregor.

I would like to thank all my friends and colleges at the University Of Huddersfield and specifically the members of the High Performance Computing Research Group: Ibad Kureshi, John Brennan, Mathew Newall, Shuo Liang and Yvonne James. I would also like to acknowledge the open-source software community, particularly: Apache (For their continued work on Hadoop) and the Linux Foundation. I would also like to acknowledge the use of the University of Huddersfield Queensgate Grid in carrying out this work.

Contents

Abstract	3
Acknowledgements	4
Contents	5
List of Figures	11
List of Tables	14
List of Algorithms	15
Abbreviations	16
1 Introduction	17
1.1 Errors In Medical Databases	17
1.2 Current Possible Solutions Using Linked Open Data	18
1.2.1 The Semantic Web and Linked Open Data	18
1.2.2 Big Data and Healthcare	19
1.2.3 Current Implementation	20
1.3 Project Aims and Motivations	21
2 Background Technologies	22
2.1 The Resource Description Framework (RDF)	22
2.1.1 An RDF Statement	23
2.1.2 RDF Graph Representation	23

2.1.3	RDF Serialisation Formats	24
2.2	Triplestores	26
2.2.1	Jena	26
2.3	Simple Protocol and RDF Query Language (SPARQL)	27
2.4	Database Joins	28
2.5	Hadoop And The Map/Reduce Programming Model	29
2.5.1	Hadoop Cluster Components	30
2.5.2	The Hadoop Distributed File System (HDFS)	30
2.5.3	Map/Reduce Programming Model	31
2.5.4	Sort/Shuffle, Partitioner and Combiner Stages	32
2.5.5	Benefits and Drawbacks	33
2.5.6	Associated Technologies	34
3	Literature Review	36
3.1	Research Fields	36
3.2	Distributed Native Triplestores	37
3.3	Performing Dataset Joins Via Hadoop	38
3.3.1	Map-Side Join	39
3.3.2	Reduce-Side Join	40
3.3.3	Cascade Join	41
3.3.4	Broadcast Join	42
3.4	Feasibility Of Using Hadoop For RDF Processing	42
3.5	Strategies For Storing RDF Data On The HDFS	43
3.6	Queries on RDF Data Using Map/Reduce	44
3.6.1	Existing Theoretical Approaches	44
3.6.2	Presented Performance Results	51
3.7	Limitations Of Existing Hadoop RDF Solutions	53
3.7.1	Hadoop Joining Strategies	54
3.7.2	Data Upload Stages	55
3.8	Possibility Of A Highly Optimised Solution	55

4	Analysis Of Current Jena TDB Implementation	57
4.1	Current Implementation	57
4.2	Limitations of Existing Solution	58
4.3	Current Implementation Performance Analysis	59
4.3.1	Upload Time	59
4.3.2	Query Time	59
4.3.3	Analysis	60
4.4	The Need For A New Framework	61
5	Hadoop Implementation and Algorithm Design	62
5.1	Structure and Distribution Of The Real-World Medical Data	62
5.1.1	Subject, Predicate and Object Distribution	63
5.1.2	Distinct Triple Group Distribution	63
5.2	Data Generation	64
5.3	Analysis Of Current SPARQL Queries	64
5.3.1	Min/Max Queries	65
5.3.2	More Complex Queries	66
5.4	Hadoop-Based Framework Design	67
5.4.1	Required Software Functionality	67
5.4.2	Why Use Hadoop As The Basis For The New Framework?	68
5.4.3	The Two Approaches Taken	68
5.5	Query Planning	69
5.6	Approach One - Data Upload Algorithm	70
5.6.1	Stage One - Compression	72
5.6.2	Stage Two - Sort On Subject	72
5.6.3	Algorithm Pseudocode	73
5.7	Approach One - Data Query Algorithm Design and Joining Strategies	73
5.7.1	Selection Stage	75
5.7.2	Join Stage	78
5.8	Approach Two	81
5.8.1	Selection Phase	81

5.8.2	Join Phase	82
5.9	Implementation Summary	83
5.9.1	Key Theoretical Performance Advantages	83
5.9.2	Summary	83
6	Test Environment and Optimisations	86
6.1	Details Of Test Environments	86
6.1.1	Single Machine	86
6.1.2	Dedicated Hadoop Cluster	87
6.2	Hadoop Cluster Performance Optimisations	87
6.2.1	Benchmark Generation	88
6.2.2	Optimum Number Of Map/Reduce Tasks	88
6.2.3	Optimum JVM Heap Memory Allocation	89
6.2.4	JVM Re-use	91
6.2.5	Sort Memory Allocation	92
6.2.6	Final Configuration Values	92
7	Results	94
7.1	Testing Methodology	94
7.2	Single Node Results	95
7.3	Single Node Results - Approach One	95
7.3.1	Upload Results	95
7.3.2	Upload Results Showing Breakdown Of Passes	95
7.3.3	Query Results	97
7.3.4	Query Results Showing Breakdown Of Passes	97
7.4	Single Node Results - Approach Two	98
7.4.1	Query Results	98
7.4.2	Query Results Showing Breakdown Of Passes	99
7.5	Single Node Comparison	99
7.5.1	Approach Comparison	99
7.5.2	Comparison With Jena TDB	101

7.6	Hadoop Cluster Results	103
7.7	Cluster Results - Approach One	103
7.7.1	Upload Performance Across Eight Nodes	103
7.7.2	Query Performance Across Eight Nodes	104
7.7.3	Performance Scalability Across Number Of Nodes	104
7.7.4	Speed-Up Analysis Of Approach One's Query Stage	105
7.8	Cluster Results - Approach Two	106
7.8.1	Query Performance Across Eight Nodes	106
7.8.2	Performance Scalability Across Number Of Nodes	108
7.8.3	Speed-Up Analysis Of Approach Two's Query Stage	108
7.9	Cluster Approach Comparison	109
7.9.1	Approach Comparison On Eight Nodes	109
7.9.2	Approach Comparison On Four Nodes	111
7.9.3	Approach Comparison With Jena	111
7.10	Naïve versus Optimised Implementation	112
8	Interpretation and Discussion Of Results	114
8.1	Summary and Significance Of Results	114
8.1.1	Single Node Results	114
8.1.2	Cluster Results	115
8.1.3	Cluster Size Versus Speed-Up Factor	115
8.1.4	Approach Comparison	116
8.1.5	Underlying Pass Comparison	118
8.2	Comparison With Literature	118
9	Conclusions And Further Work	121
9.1	Conclusions	121
9.1.1	Project Summary	121
9.1.2	Aims and Objectives Achieved	122
9.1.3	Evaluative Conclusions	122
9.2	Further Work	124

9.2.1	Improvements To The Project	124
9.2.2	Expansion Of The Project	125
9.3	Final Conclusions	126
References		128
 A Appendix - SPARQL Queries		134
A.1	Original SPARQL Queries	134
 B Appendix - Hadoop Cluster Configuration Files		138
B.1	HDFS Site	138
B.2	Mapred Site	138
B.3	Core Site	139
 C Appendix - Approach One Source Code		140
C.1	Upload Algorithm	140
C.1.1	Map 1 - Compressor	140
C.1.2	Map 2 - Create Single Line On Subject	142
C.1.3	Reduce 1 - Create Single Line On Subject	143
C.2	Query Algorithm	143
C.2.1	Map 1 - Pass 1	143
C.2.2	Reduce 1 - Pass 1	146
C.2.3	Map 2 - Pass 2	149
C.2.4	Reduce 2 - Pass 2	150
 D Appendix - Approach Two Source Code		158
D.1	Query Algorithm	158
D.1.1	Map 1 - Pass 1	158
D.1.2	Reduce 1 - Pass 1	161
D.1.3	Map 2 - Pass 2	165
D.1.4	Reduce 2 - Pass 2	165

List of Figures

1.1	The Semantic Web Stack	18
1.2	Real World Open Linked Data	19
2.1	A Sample RDF Dataset	24
2.2	Graphic Representation Of RDF Data	24
2.3	Basic Map/Reduce Workflow	31
2.4	Full Map/Reduce Workflow	33
3.1	SPARQL Map/Reduce Research Areas	36
3.2	Map-Side Join	39
3.3	Reduce-Side Join	40
3.4	Kulkarni's Selection Phase	46
3.5	Kulkarni's Join Phase	47
3.6	Mazumdar's Example SPARQL Query	47
3.7	Mazumdar's Query Execution Plan	48
3.8	Mazumdar's First Join	48
3.9	Goasdoue's 1-Clique Query	50
3.10	Goasdoue's Central Clique Query	50
3.11	Mazumdar's Performance Results	51
3.12	Husain's Performance Results	52
3.13	Goasdoue's Upload Performance Results	53
3.14	Goasdoue's Query Performance Results	53
4.1	Current Approach For Checking Medical Data	58
4.2	Jena TDB Upload Performance	60

4.3	Jena TDB Query Performance	60
5.1	Triple Group Joining Plan	71
5.2	Approach One - Selection Stage Workflow	75
5.3	Map-Side Join Example	76
5.4	Reduce-Side Join Example	77
5.5	Join Stage Workflow	79
5.6	Hadoop Job Output Showing Counters	81
5.7	Approach Two - Selection Stage Workflow	82
6.1	Number of Map/Reduce Tasks Results	89
6.2	Hadoop Memory Footprint Calculation	90
6.3	Optimum Heap Size Result	90
6.4	JVM Reuse Result	91
6.5	Hadoop Sort IO Result	92
7.1	Approach One - Single Machine Upload Time	96
7.2	Approach One - Single Machine Upload Time Showing Breakdown Of Passes	96
7.3	Approach One - Single Machine Query Time	97
7.4	Approach One - Single Machine Query Time Showing Selection And Join Stages	98
7.5	Approach Two - Single Machine Query Time	99
7.6	Approach Two - Single Machine Query Time Showing Pass Breakdown	100
7.7	Single Node Query Performance Comparison Between Approaches	101
7.8	Single Node Total Performance Comparison Between Approaches	101
7.9	Single Node Total Performance Comparison Between Hadoop And Jena	102
7.10	Approach One - Upload Performance Across 8 Nodes	103
7.11	Approach One - Query Performance Across 8 Nodes	104
7.12	Approach One - Upload Performance Scalability Across Nodes	105
7.13	Approach One - Query Performance Scalability Across Nodes	106
7.14	Approach One - Query Speed-Up Factor	107
7.15	Approach Two - Query Performance Across 8 Nodes	107

7.16 Approach Two - Query Performance Scalability Across Nodes	108
7.17 Approach Two - Query speed-up Factor	109
7.18 Eight Node Cluster Approach Query Comparison	110
7.19 Eight Node Cluster Approach Total Comparison	110
7.20 Four Node Cluster Approach Total Comparison	111
7.21 Hadoop Approaches Versus Jena	112
7.22 naïve versus Optimised Implementation	113

List of Tables

2.1	Join Example 1	29
2.2	Join Example 2	29
2.3	Join of Table 1 and 2 on Course-ID	29
5.1	Subject, Predicate and Object Distribution	63
5.2	Triple Group Distribution	63
6.1	Specification Of The Single Machine	86
6.2	Specification Of The Hadoop Cluster	87
6.3	Hadoop Configuration Files	88
6.4	Hadoop Memory Allocation on Cluster	91
6.5	Hadoop Cluster Final Configuration Values	93
8.1	Specification Of The Clique-Square Hadoop Cluster	119

List of Algorithms

1	Medical Data Upload Algorithm	74
2	Medical Data Query Map/Reduce Iteration 1	78
3	Medical Data Query Map/Reduce Iteration 2	85

Abbreviations

RDF	R esource D escription F ramework
SPARQL	S imple P rotocol A nd R DF Q uery L anguage
URI	U niform R esource I dentifiers
URL	U niform R esource L ocator
HDFS	H adoop D istributed F ile S ystem
JVM	J ava V irtual M achine
RAM	R andom A ccess M emory
BGP	B asic G raph P atern
NHS	N ational H ealth S ervice
LUBM	L ehigh U niversity B enchmark

Chapter 1

Introduction

1.1 Errors In Medical Databases

Recent technological advances in modern healthcare have led to a vast wealth of patient data being collected. This data is not only utilised for diagnosis but also has the potential to be used for medical research. For any conclusions from this research to be of worth, the underlying data needs to be of high quality. However, according to Goldberg, Niemierko, and Turchin (2008), there are often many errors in datasets used for medical research. In this study they found error rates ranging from 2.3% to 26.9% in a selection of medical research databases. Salati et al. (2011) highlight a series of metrics on which the quality of medical data can be judged. These are accuracy, completeness, consistency and believability. Salati et al. (2011) state that the errors which they find present in medical databases can have three different etiologies: firstly, errors which were present in the original data and then copied into the medical databases; secondly, errors which occur due to misinterpretation of the original data; finally, errors which occur when the data is being entered into the database. With such high error rates present in medical databases there is clear need for a system to assess the quality of data before it is used as the basis of cutting edge research.

1.2 Current Possible Solutions Using Linked Open Data

There has been an attempt to create a system which automatically assesses a given dataset to check for errors contained within (Corsar, Moss, & Piper, 2012). Such a system has been developed using Semantic Web and Linked Open Data technologies, both of which will be expanded upon in this section. The concepts relating to the emerging field of Big Data and how this relates to healthcare will also be explored.

1.2.1 The Semantic Web and Linked Open Data

The original vision of the Semantic Web was established by Berners-Lee, Hendler, and Lassila (2001). The vision was to create a machine-readable structure to supplement the pre-existing human-readable content of web pages. According to Davies, Fensel, and van Harmelen (2003) this structure consists of meta-information, defining specific properties about any given element of data. This meta-information captures some of the meaning, or semantics behind the data, leading to the concept being named the Semantic Web (Antoniou & Harmelen, 2008). The Semantic Web is implemented in a series of layers, which collectively form the Semantic Web stack illustrated in Figure 1.1. A more in-depth view of one of the fundamental layers of the Semantic Web, the Resource Description Framework (RDF), can be found in Chapter 2.

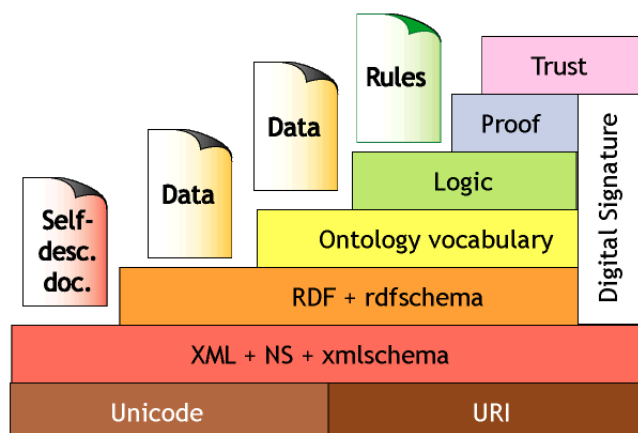


FIGURE 1.1: The Semantic Web Stack (W3C, 2001).

Linked Open Data is defined by Bizer, Heath, and Berners-Lee (2009) as being simply different sources of data which are inter-linked together via the web. These data sources can be geographically distributed and vary in source, content and format. To enable these datasets to be linked, Semantic Web technologies and concepts are utilised. The number of real-world datasets linked to others via the Open Linked Data philosophy is increasing rapidly as Figure 1.2 demonstrates.

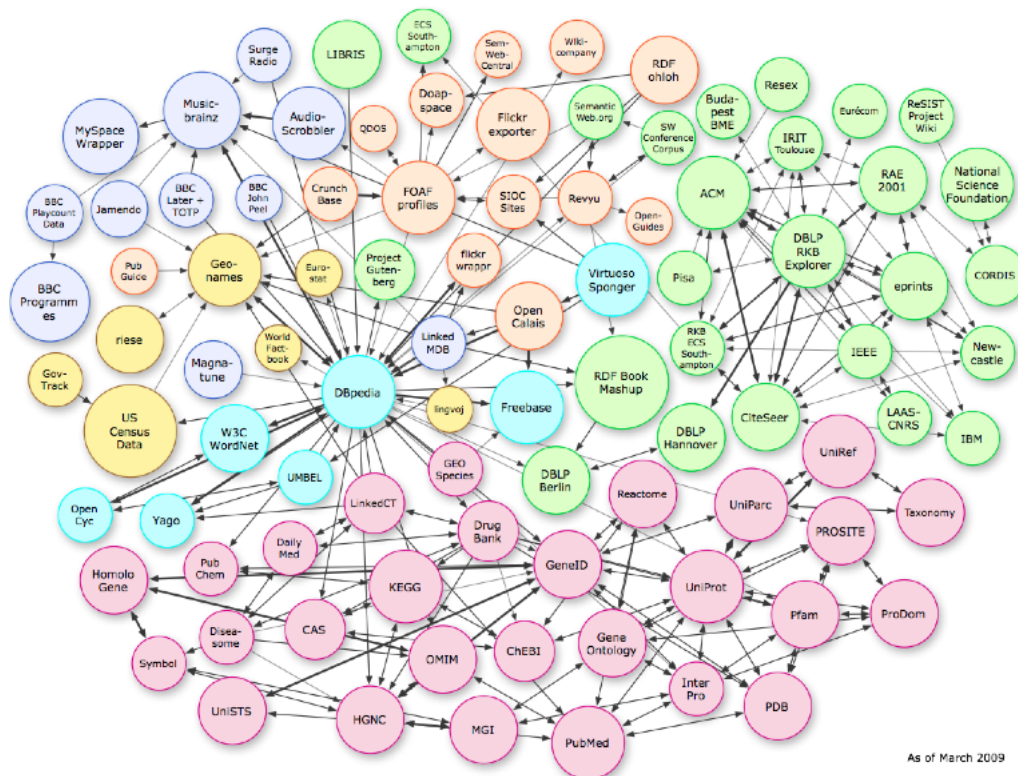


FIGURE 1.2: Real World Open Linked Datasets (Bizer et al., 2009).

1.2.2 Big Data and Healthcare

Big Data is a rapidly increasing area of interest within the computer science field and draws upon areas from many different disciplines. Both academic and industrial fields are generating and collecting data at an unprecedented rate and scale. This data, and the analysis of it, is being used to replace models and guesswork as the basis for decision-making Agrawal et al. (2011). While there have been some early successes, the true potential of Big Data and the decisions which can be made from its analysis have, according to Agrawal et al.

(2011), have not yet been fully realised. This is due to the technical challenges which storing and processing the massive volume of data present. Gartner (2011) argues that it is not only the volume of data which is challenging current technology, but also the variety (the heterogeneity of data, representation, and semantic interpretation) and velocity (data arrival and required processing rate) of the data.

The biomedical and general healthcare fields have the potential to be one of the biggest contributors to and benefactors from the big data phenomenon. Feldman, Martin, and Skotnes (2012) state that the volume of worldwide healthcare data as of 2012 is estimated to be over 500 petabytes. This figure is predicted to increase by a factor of 50 to a value of 25,000 petabytes by 2020. Agrawal et al. (2011) state that one of the biggest contributors to this data explosion will be the use of continuous monitoring machines, both in hospitals and at home. Howe et al. (2008) explain that analysing this data has the potential to alter biomedical research and significantly improve healthcare diagnosis.

1.2.3 Current Implementation

A Linked Data approach to assessing the quality of medical data has been created. Corsar et al. (2012) state that this framework can be broken down into three key stages. Firstly, pre-existing medical data is converted into RDF data and stored in Jena TDB. Secondly, the data can be annotated with provenance information, such as the specification of the machines which recorded the data. Lastly, a data checking component assesses the quality of the data via a series of SPARQL rules. A more detailed look at the current implementation can be found in Chapter 4. The current implementation relies on traditional semantic web technologies which are not well-adapted to process the volume of data which healthcare entails. Due to this, the current framework suffers from performance and scalability issues when processing massive RDF datasets.

1.3 Project Aims and Motivations

Due to the potential volume of RDF-based medical records, performance issues with the current framework could limit its adoption in the wider medical community. This project will explore alternative methods for both storing medical RDF data and also assessing its quality via a series of SPARQL queries. One of the key considerations for this project will be performance and scalability of data upload and also query response time.

The specific aims and objectives of this project are as follows:

- i) To create a new storage and query framework for medical data, with specific emphasis being placed on performance. This will be implemented in Apache Hadoop Map/Reduce.
- ii) To test the newly developed Hadoop framework using real-world medical data.
- iii) To compare performance and scalability of this Hadoop framework against the existing implementation of the Linked Data approach to assessing the quality of medical data and draw relevant conclusions.

Chapter 2

Background Technologies

In this chapter the fundamental, underlying technologies utilised in this project will be introduced and explained. These technologies include RDF, SPARQL, Triplestores, Hadoop and the Map/Reduce programming model.

2.1 The Resource Description Framework (RDF)

The Resource Description Framework (RDF) is one of the fundamental layers of the semantic web stack illustrated in Figure 1.1. According to W3C (2004) RDF is a language specifically designed to express the relationship between elements on the world wide web. The philosophy behind RDF is about making statements that are easy for machines or computer programs to understand and process.

Antoniou and Harmelen (2008) state that the fundamental concepts of RDF can be considered to be: Resources, Properties and Statements.

- Resources are objects or things, about which there is a need to express some information. Resources are represented by Uniform Resource Identifiers (URI) which is a unique way of identifying a specific Resource.
- Properties describe the relationship between different Resources. Each Property is also identified via a URI.

- Statements are the complete RDF triple. An RDF triple consists of a Resource, a Property and a Value. Values can be a Resource or literals (such as a string or integer).

2.1.1 An RDF Statement

Ogbuji (2000) states that an RDF Statement comprises three distinct structural components: a Subject (about which the statement is being made), a Predicate (describing the relationship between Subject and Object) and an Object (an attribute of the Subject). Ogbuji (2000) expands upon the concept of an RDF Statement by drawing comparisons with the English language. For example the sentence '*John Brennan is a lecturer at the University of Huddersfield*' can be represented in RDF with the following structure:

- Subject ('John Brennan')
- Predicate ('lecturer at')
- Object ('University of Huddersfield')

Resources in RDF, as previously discussed, are represented by URI's. A Uniform Resource Locator (URL) is a subset of URI's which is often used to locate an RDF Resource. Therefore the statement from earlier could more accurately be represented as:

- Subject (http://www.hud.ac.uk/staff/John_Brennan)
- Predicate (<http://www.hud.ac.uk/role/lecturer>)
- Object (<http://www.hud.ac.uk>)

2.1.2 RDF Graph Representation

As RDF is used to express the relationship between resources, it is often represented as a graph. For example the RDF graph displayed in Figure 2.2 shows the relationships between different elements from the dataset displayed in Figure 2.1. RDF graphs are extremely

useful in visualising how simple RDF triple statements can be built up to express complex relationships between numerous unique resources.

```
:stud1 :takesCourse :db      :stud1 :member :dept4
:stud2 :takesCourse :os      :stud2 :member :dept1
:prof1 :advisor :stud1      :prof2 :advisor :stud2
:prof1 :name "bob"          :prof2 :name "alice"
:stud1 :name "ted"          :dept1 rdf:type :Dept
```

FIGURE 2.1: A Sample RDF Dataset (Goasdoué et al., 2013).

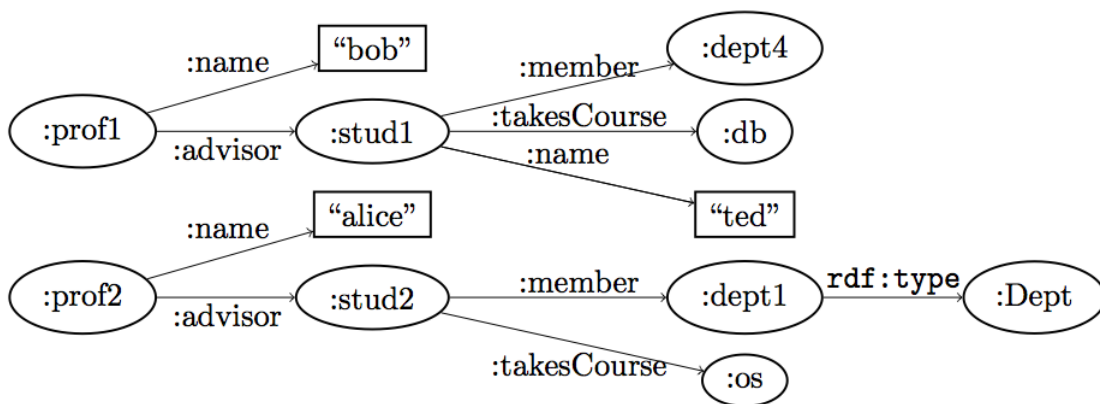


FIGURE 2.2: Graphic Representation Of Sample RDF Data (Goasdoué et al., 2013).

2.1.3 RDF Serialisation Formats

Segaran, Evans, and Taylor (2009) explain that there are several formats in which an RDF statement can be serialised, including RDF/XML, N3 and N-Triple. The original RDF serialisation is RDF/XML, which builds on the XML structure of tags to represent a Triple. The earlier statement would be expressed in RDF/XML as:

```
<?xml version="1.0" encoding="UTF-16"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns"
  xmlns:unidomain="http://www.hud.ac.uk/my-rdf-ns">

  <rdf:Description rdf:about="http://www.hud.ac.uk/staff/John_Brennan">
    <unidomain:lecturerAt rdf:resource="Univerity-of-Huddersfield"/>
```

```
</rdf:Description>
</rdf:RDF>
```

The N-Triple notation is a simpler way of representing an RDF Triple. Each line of N-Triple contains a single RDF Statement in full, with the subject, predicate and object separated by whitespace. Both Subject and Object are represented as a full URI enclosed by angle brackets. The earlier statement would be expressed in N-Triple as:

```
<http://www.hud.ac.uk/staff/John_Brennan> <http://www.hud.ac.uk/role-
/lecturer> <http://www.hud.ac.uk>
```

While the structure of N-Triple is extremely simple, it can lead to a lot of repetition with larger RDF datasets. It is common for RDF datasets to contain numerous statements that share at least one element, which the N-Triple format will repeat. The N-Triple format also expresses RDF data with the full URI, again a wasteful method.

The N3 format attempts to rectify some of the inherent inefficiencies with the N-Triple format. It does this via two methods. Firstly, N3 replicates the XML namespace mechanism to allow definition of URI prefixes at the beginning of a RDF document. Secondly, N3 provides a method of representing statements that share a common element. For example the statement from earlier could be expanded on in N3 as:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix hud: <http://www.hud.ac.uk/#>.

hud:John_Brennan rdf:type foaf:person;
    hud:lecture hud:uni.
```

Here an extra triple has been introduced. As it shares a common subject with the original statement, N3 does not repeat it; instead it uses a semi-colon to represent a shared element.

2.2 Triplestores

A triplestore is a framework designed especially to provide persistent storage of and the ability to query RDF data Haslhofer, Roochi, Schandl, and Zander (2011). According to Sequenda (2013), triplestores can be categorised into three groups based on the storage implementation. These groups are Native, RDBMS-backed and NoSQL. Native triplestores are designed specifically for, and thus optimised for, storing RDF data. These Native implementations are independent of other Relational Database Management Systems (RDBMS) and store the RDF data in the local filesystem. RDBMS-backed triplestores store the RDF data in a traditional relational database such as MySQL. NoSQL triplestores are a recent development designed to exploit the advances being made in NoSQL databases such as Cassandra.

According to Weaver and Williams (2009) while many of the common native triplestores use advanced database techniques to enable fast querying of RDF data, they require a lot of pre-processing which can often lead to a prohibitively long data upload time.

2.2.1 Jena

Jena is one of the mostly widely utilised tools used to store and process RDF data and was originally developed by Hewlett-Packard (Wilkinson, Sayers, Kuno, & Reynolds, 2003). It comprises a Java API, allowing users to write programs that create or manipulate RDF data, as well as a native (Jena TDB) and also a RDBMS-backed (Jena SDB) triplestore to house and query RDF data. Jena implements a query engine called ARQ to perform SPARQL queries on the triplestore.

While Jena is extremely popular, Husain, Doshi, Khan, and McGlothlin (2009) state that it suffers from performance issues, particularly related to the initial upload of RDF data. Also due to the fact that Jena is limited to running on just a single machine, it can only process a relatively small number of triples at a time. Husain et al. (2009) report that a machine with 2GB of system memory could only store a dataset of 10 million triples.

2.3 Simple Protocol and RDF Query Language (SPARQL)

According to DuCharme (2011) the Simple Protocol and RDF Query Language (SPARQL) provides a way of querying RDF data stored in a triplestore and is the W3C standard for doing so. SPARQL can be compared to the Structured Query Language (SQL), used to query standard relational databases, as they share a similar syntax and logic. Antoniou and Harmelen (2008) expand on SPARQL by explaining that conceptually, it is based around matching graph patterns. According to W3C (2013), a query evaluates what is known as a basic graph pattern (BGP). Each BGP can contain several query patterns all resembling an RDF triple. Variables are able to substitute any part of the BGP to allow individual elements to be extracted from any input triples matching the rest of the pattern (W3C, 2013). Goasdoué et al. (2013) state that the normative syntax for a SPARQL query is:

```
SELECT ?v1...?vn WHERE {t1...tn}
```

with the `SELECT ?v1...?vn` element representing distinct variables occurring in the input data and the order in which they are to be returned. `FROM` is an optional element not shown, that can be utilised to determine the data source to be queried. The `WHERE t1...tn` element forms the BGP and is a series of triple query patterns that will match the graph of any input triples against the ones specified in the BGP.

A basic SPARQL query that selects all lecturers at the University of Huddersfield would be:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX hud: <http://www.hud.ac.uk/#> .
```

```
SELECT ?name
WHERE
{
    ?name hud:Lecturer hud:Uni .
}
```

This simple query would return a list of all elements (in this case lecturers at the University) that matched the triple graph pattern, bound to the variable `?name`. However, often more advanced information is required from a query. The following query would select all people who are lecturers at the University of Huddersfield and are also part of the School of Computing:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX hud: <http://www.hud.ac.uk/#> .
```

```
SELECT ?name
WHERE
{
    ?name hud:Lecturer hud:Uni .
    ?name hud:School hud:Computing .
}
```

In this example the BGP has two query patterns to match, both of which share a common variable. According to DuCharme (2011), replicating a variable between two triple patterns in a SPARQL query is done to join together the two patterns. When the SPARQL query engine finds a triple with the predicate of `hud:Lecturer` and an object of `hud:Uni` it would bind the subject of such a triple to the variable `?name`. When processing the second pattern, possible matches will be limited to results containing the variable `?name`. Effectively results from the first pattern will be joined to results from the second so that only elements meeting both conditions will be returned. The process of joining data comes from the world of relational databases and is explored in the next section.

2.4 Database Joins

Silberschatz, Korth, and Sudarshan (2011) explain that theory behind database joins is structured around relational algebra. They expand this point by stating that relational algebra encompasses a set of operations on relationships which mirror standard algebraic operations. While standard algebra includes addition, multiplication and subtraction, which starts with a set of numbers as input and returns a number as output, relational algebra operations takes a set of relations for input and returns a new relation as output. The most common relational algebra operation used when discussing joins is the Natural Join (Represented by \bowtie) (Silberschatz et al., 2011). The Natural Join is the combining of two datasets by merging matching elements based on a common key, called a Foreign Key (Miner & Shook, 2012). All patterns that match are included in the result of the join.

Table 2.1 and Table 2.2 contain data that will be joined via a natural join. In this example the values of the Teaches-Course from table 2.1 will be joined to 2.2 via the values from the Course-ID.

Name	School	Teaches-Course
John	Engineering	165
Matt	Computing	132
Ibad	Engineering	50
Yvonne	Computing	74

TABLE 2.1: Join Example 1

Course-ID	Course-Name
165	DSP
50	PCA
74	CS

TABLE 2.2: Join Example 2

The join can be expressed as the following: $(TeachesCourse \bowtie CourseID) \rightarrow NewTable$. Joining the two datasets will create a new dataset shown in Table 2.3. All results that can join via the foreign key will be included in the table, while results not matching (for example the record for 'Matt') will be absent.

Name	School	Teaches-Course	Course-Name
John	Engineering	165	DSP
Ibad	Engineering	50	PCA
Yvonne	Computing	74	CS

TABLE 2.3: Join of Table 1 and 2 on Course-ID

2.5 Hadoop And The Map/Reduce Programming Model

Lam (2010) defines Hadoop as an open source framework for the storing and processing of internet-scale data in a distributed manner. Hadoop tackles the problem of 'Big Data' by distributing the storage and processing of data to numerous machines. White (2010) states that Hadoop comprises two main components: the Hadoop Distributed File System (HDFS), used for storing data across a Hadoop cluster and the Map/Reduce programming framework, used to process the data. Hadoop has emerged as the de-facto standard for the processing of Big Data introduced in section 1.2.2.

2.5.1 Hadoop Cluster Components

According to Chandar (2010) a Hadoop cluster consists of the following components, all of which are Java Virtual Machines (JVM):

- *JobTracker* - Master node which controls the submission and scheduling of jobs on the cluster.
- *NameNode* - Controller node for the HDFS, which keeps track of which node stores what data.
- *TaskTracker* - These nodes are worker nodes for Map/Reduce and are where the processing happens. A TaskTracker demon does not itself run jobs, instead it controls the spawning of separate JVMs for each Map/Reduce function.
- *DataNode* - These nodes comprise the rest of the HDFS and house the data. Usually DataNodes are also TaskTrackers, so data can be processed on the same node in which the data resides.

2.5.2 The Hadoop Distributed File System (HDFS)

Rajaraman and Ullman (2012) state that HDFS is an open source imitation of the original Google File System (GFS). Rather than being stored as a single entity, files stored on a HDFS are divided into blocks, usually 64 megabytes in size. As the HDFS is designed to be hosted via commodity machines, each block is replicated three times, with each copy being stored on a different machine. This inherent file redundancy means that node failure has no impact on cluster functionality and is another key advantage of Hadoop. Due to the block nature of HDFS, it is not suited for the storing of files smaller than the size of the block (Rajaraman & Ullman, 2012).

2.5.3 Map/Reduce Programming Model

Hadoop's Java implementation of the Map/Reduce programming model is based on Google's proprietary system introduced in a 2004 paper (Lam, 2010). According to Dean and Ghemawat (2004) a standard Map/Reduce pass consists of two distinct phases: a Mapper and a Reducer. All inputs to a Map/Reduce task are key/value pairs, with all intermediate and final output also represented in this way.

Dean and Ghemawat (2004) explain the role of each phase in the following manner: the Map function processes the input key/value pairs, performs a user-defined algorithm, then outputs an intermediate set of key/value pairs. These intermediate results are grouped on the key to ensure that all values associated with that key are sent to the same Reduce function. The Reduce function then performs the final processing and outputs the last set of key/value pairs. The basic workflow of Map/Reduce is demonstrated in Figure 2.3.

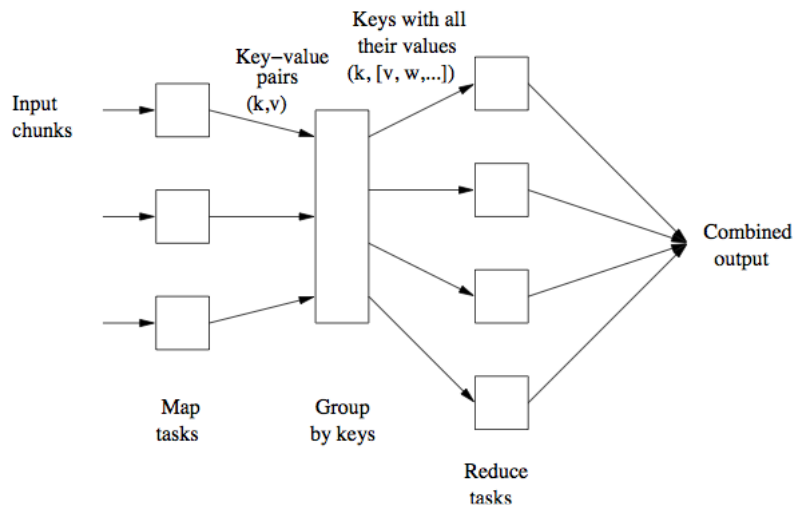


FIGURE 2.3: Simplified Map Reduce Workflow (Rajaraman & Ullman, 2012).

Rajaraman and Ullman (2012) use a word count program as a method of demonstrating how Map/Reduce operates conceptually. For this example, the input to the program will be a collection of documents stored on the HDFS. The Map function processes the documents and breaks them into a sequence of individual words. The function would then emit its intermediate key/value pair with the word being the key and the value being one. So the output would be - $(w_1, 1), (w_2, 1), \dots, (w_n, 1)$. At the grouping stage any replicated keys would have their values collated before being passed onto the Reduce function. So the input to the

Reduce function would be (w1, [1 1 1 1 1]), for a word that had five instances in the original input files. The Reduce function then would sum the values for the key and emit the total, along with the word as the final output - (w1, 5).

The pseudocode for the word-count program would be:

Map:

```
void Map(string document) {  
    for each word w in document {  
        Emit_Intermediate(w, "1");  
    }  
}
```

Reduce:

```
void Reduce (string word, list<string> values) {  
    int count = 0;  
    for each v in values {  
        count += StringToInt(v);  
    }  
    Emit_Final(word, count);  
}
```

2.5.4 Sort/Shuffle, Partitioner and Combiner Stages

Thus far the Map/Reduce model discussed has been a simplified one, with several key stages absent. These stages include Sort/Shuffle, Partition and Combiner. Together they are what manage the crucial transfer of data from Mapper to Reducer. White (2010) states that the Sort/Shuffle stage is what sorts the numerous different values emitted from the Map stage by their key, and then passes them over the network to a Reduce function. The Partitioner is responsible for deciding how the output from a Map task is divided before being sent to a Reduce task. By default Hadoop will equally divide the intermediate Map output between the number of requested Reduce tasks, while also ensuring that all the values of a certain key are located within the same partition. However, a user can write their own Partition function if they require different processes to be performed upon the intermediate data (White, 2010). Lin and Dyer (2010) state that Combiners are an optimisation that enable intermediate results to be processed before the Shuffle and Sort phase. A Combiner can be considered a mini-reducer that process the output from each Map task before it is passed to the final Reducer. As a Combiner only operates on a single Map task, it has no

guarantee of access to all values for a specific key so they are only really utilised to reduce the amount of data to be passed to the final Reducer (Lin & Dyer, 2010).

The location of these additional stages in the Map/Reduce workflow can be seen in Figure 2.4.

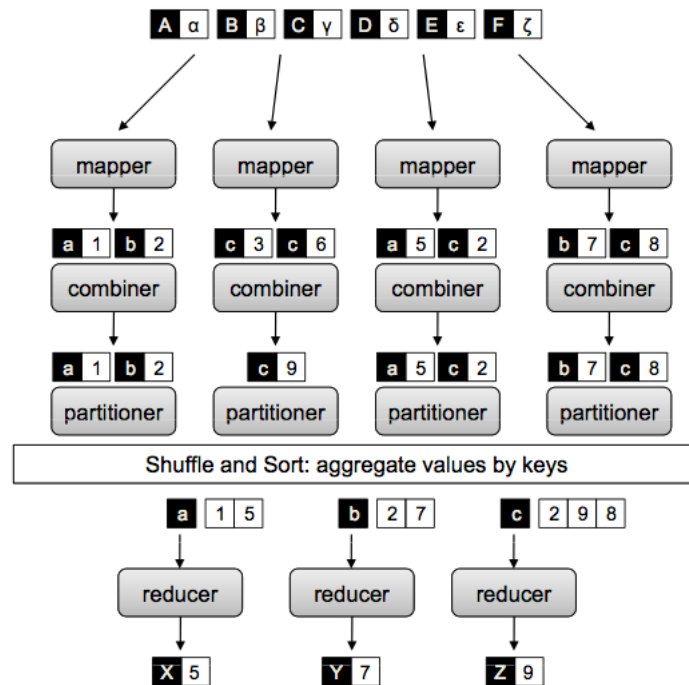


FIGURE 2.4: The Full Map Reduce Workflow (Lin & Dyer, 2010).

2.5.5 Benefits and Drawbacks

The word count example discussed in the previous section demonstrates the advantages of using the Map/Reduce programming model. Rajaraman and Ullman (2012) argue that because the Map function performs an identical algorithm on all input data, multiple instances of the Map function can process different parts of the dataset, on different machines, simultaneously; thus processing the complete datasets an order of magnitude faster than processing the datasets sequentially. Further, White (2010) argues that Hadoop's approach to large scale-data processing differs from the traditional High Performance Computing (HPC) approach. HPC compute nodes usually have no access to data locally, instead they access it via a centralised shared filesystem over a network. When processing gigabytes of data,

the network bottleneck can seriously limit the performance of the system. One of the major advantages of the Hadoop framework is that it tries to ensure that the data to be processed is available on the local drive of the compute node. This bypasses the need for a time-consuming network transfer of the data. This key feature, known as data locality, is one of the main drivers of Hadoop's performance advantage (White, 2010).

Hadoop imposes strict limits on both the Map and Reduce functions to enable it to perform the parallelisation. Map functions inherently have to work independently on one set of key/-value pairs and have no access to other pairs. The Reduce function can only access those values associated with the key used to generate the reduce function, making features such as dataset joins complicated (White, 2010). Hadoop is implemented in Java, with each of the cluster components discussed in section 2.5.1 running as a separate JVM. During a standard Map/Reduce iteration each Map and Reduce task incurs a start-up penalty associated with spawning the JVM. Due to this penalty it is usually best to limit the number of passes required to complete a set task (Joshi, 2012). Another disadvantage that is associated with Hadoop is the steep learning curve in discovering how to correctly program a Map/Reduce job. White (2010) argues that creating a new Map/Reduce task even for an experienced programmer is not a quick process due to the complexity of the code and number of required functions for even a simple job.

2.5.6 Associated Technologies

There are a selection of other technologies under the Apache Hadoop banner which build extra functionality onto both the HDFS and Map/Reduce layers. These technologies include: H-Base, Hive and Pig.

H-Base is an open-source implementation of Google's Big Table design (Chang et al., 2006). Taylor (2010) states that it provides a structured, fault tolerant and scalable database layer that resides on top of the HDFS. Unlike many other projects, H-Base enables real-time random read and write access to the data contained within a table. H-Base provides no query language, so users must access the data via a standard Map/Reduce job.

Hive is a project created by Facebook to provide not only a structured data store, but also a way of querying the data via an SQL-like language called Hive QL (White, 2010). According to Taylor (2010) a query created using Hive QL is automatically translated into the required number of Map/Reduce tasks, enabling users who are familiar with SQL to easily start processing data via Hadoop. Hive will automatically perform any requested joins between data elements. However White (2010) shows that if multiple joins are required across data elements residing in different columns, Hive will perform an un-optimised and slow series of cascade joins (Cascade joins will be discussed further in section 3.3.3).

According to Taylor (2010), Pig was created to be a high-level data-flow language, comprising two elements: The Pig data-flow language named Pig-Latin and the Pig execution environment which enables the job to run on a local or Hadoop distributed environment. Taylor (2010) further explains that the Pig-Latin language allows users to express their jobs as a series of operations and transformations to a selection of input data to create output. These operations include filtering, sorts and joins. A standard Pig-Latin job usually contains only a small selection code to complete comparatively complex tasks. In a similar fashion to Hive, Pig automatically translates these operations into a series of Map/Reduce iterations. While Pig makes creating Map/Reduce tasks simpler, the trade-off is that a natively written job will have comparatively increased performance, especially when performing joins requiring multiple passes (White, 2010).

Chapter 3

Literature Review

3.1 Research Fields

An analysis of literature from several different fields is required in order to successfully and efficiently devise a method to perform SPARQL queries via Map/Reduce. Figure 3.1 effectively demonstrates the overlap between the different fields from which this project draws. This chapter will explore prior research conducted into area 2 highlighted in Figure 3.1: performing SPARQL queries using Map/Reduce.

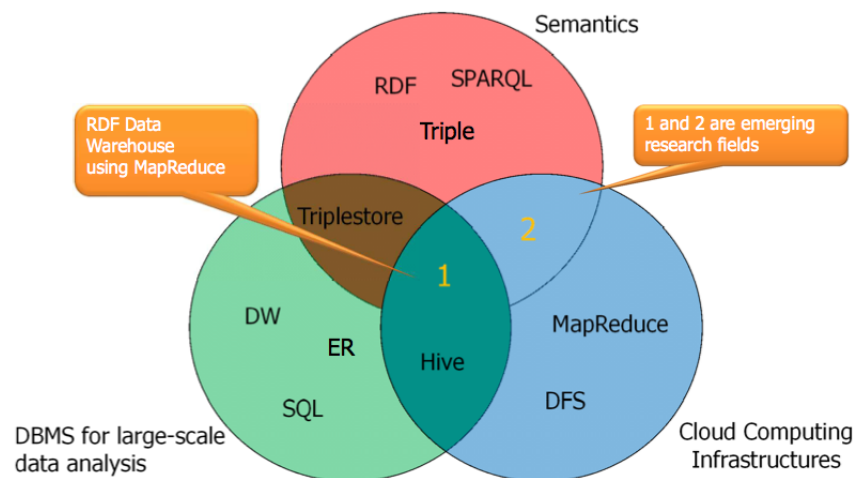


FIGURE 3.1: SPARQL Map/Reduce Research Areas (Myung, 2010).

3.2 Distributed Native Triplestores

According to Owens, Seaborne, and Gibbins (2008) the volume of structured RDF data is quickly outstripping the ability of native triplestores to store and query. Many of the most commonly used triplestores are designed to only utilise a single machine. However, in an effort to improve both upload and query performance, there have been attempts to create distributed native triplestores that either augment or replace existing solutions.

Weaver and Williams (2009) developed a system that is able to process and query RDF data on a selection of HPC machines, ranging from commodity Beowulf clusters to IBM Blue Gene machines. Their implementation is created using the Message Passing Interface (MPI) to enable parallelisation and requires no processing of the data to be performed before it is stored. Weaver and Williams (2009) state that while their system performs well when compared with traditional triplestores, it requires that the entire RDF data set, along with all intermediate results, fit in the main system memory. They highlight that this approach has two clear disadvantages. Firstly, any machine must have an equal or greater amount of RAM to the size of the RDF dataset. This requirement would place a finite limit on dataset size as it could never exceed the size of system memory. Secondly, as RAM is volatile storage, this would not be an ideal solution to permanently house the data, as a hardware or power failure would result in complete data loss.

Owens et al. (2008) present a distributed triple store designed as a direct replacement for Jena TDB. Their implementation draws on many of the same optimisations employed to make traditional databases efficient in a parallel environment, while also keeping specific RDF processing benefits associated with the existing Jena TDB system. The Clustered TDB divides the underlying hardware into one of two possible roles; Query Coordinators which receive queries and store the location of the data on the rest of the cluster or Data Nodes which store the data and perform any operations such as sorts or joins. Owens et al. (2008) conclude that when the clustered system is compared with the standard Jena TDB, it shows increased performance in triple load rates. However the system does not scale well when performing the joins required to complete a SPARQL query, with many of the performed tests showing a sharp drop-off in performance when running on a cluster

of three machines. On one such query, a Clustered TDB system comprising three nodes takes four times the amount of time to complete the query, compared with a standard TDB system.

Harris, Lamb, and Shadbolt (2009) introduce a distributed triplestore called 4-Store. 4-Store is designed as a complete implementation of a triplestore, handling both the storage and query requirements. A 4-Store cluster consists of a single processing node controlling a selection of data nodes on which the RDF data is stored. Two studies, one conducted by Patchigolla (2011) and one by Haslhofer et al. (2011) both find 4-Store to be one of the fastest from a selection of different triplestores. Haslhofer et al. (2011) show that even running on a single node 4-Store is often twice as fast as Jena TDB. However 4-Store has shown that query performance does not scale well, with performance remaining static once the cluster size is increased past 4 nodes (Patchigolla, 2011).

3.3 Performing Dataset Joins Via Hadoop

As a complex SPARQL query requires multiple joins, any implementation designed to process them via Map/Reduce also requires the ability to perform joins. Joins in SPARQL are handled by the underlying engine (such as Jena's ARQ) and therefore are seamless to the user. However according to Miner and Shook (2012), performing joins in a Map/Reduce environment is potentially the most complex operation to achieve efficiently. Map/Reduce was designed to process large datasets by looking at each element in isolation and processing it sequentially, so joining two potentially massive datasets is beyond Hadoop's design paradigm.

While joins in Map/Reduce are complicated, several different strategies have emerged that make them possible. These join strategies include: Map-Side, Reduce-Side, Broadcast and Cascade joins (Chandar, 2010). These strategies will be explored in greater detail below. For the Map-Side and Reduce-Side joins, both a Two-way (joining two items in a single Map/Reduce iteration) and Multi-way join (Joining multiple items in a single Map/Reduce iteration) will be considered.

3.3.1 Map-Side Join

According to White (2010) a Map-Side join functions by joining the datasets before they reach the Map function. This eliminates the need for the datasets to be passed to the Reduce function, as the output of the Map function is the final joined output. While avoiding the Reduce phase means that a Map-Side join is potentially the quickest join method, the datasets to be joined have to meet very strict formatting conditions for it to work. White (2010) expands on two of the fundamental formatting conditions: firstly all input data must be partitioned using the partitioner, with the number of partitions in each input dataset being identical. Secondly, all input data must be sorted via the same key to be used for the join, with all values of a certain join key residing in the same partition. According to Palla (2009) the Map-Side join uses this precise structure of the input data to enable joining without data being passed to the reduce function. Figure 3.2 shows a Map-Side join workflow.

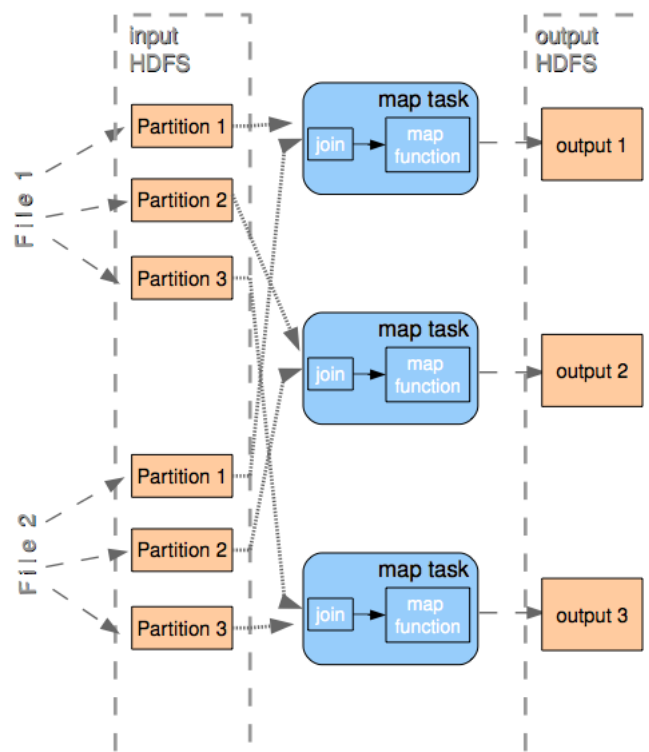


FIGURE 3.2: Map-Side Join (Palla, 2009).

The formatting conditions required for a Map-Side join often mean that for many datasets stored via their original structure, using a Map-Side join is not practical (Chandar, 2010).

However with the introduction of a pre-processing stage, comprising a single Map/Reduce iteration which simply passes the data through the framework, it can be possible to format the input data so that it is compliant with the Map-Side join input requirements (Palla, 2009).

3.3.2 Reduce-Side Join

The Reduce-Side join is a more common approach utilised to join data via Map/Reduce and is implemented using the complete Map/Reduce iteration (Palla, 2009). According to White (2010) the basic workflow for the Reduce-Side join is as follows: Firstly the Map function iterates through all records, tagging them with their source dataset and sets the Map output key as the join key. This will ensure that all values featuring that key are sent to one Reduce function. The Reduce function can then join the required elements and emit the final output. A simple two-way Reduce-Side join workflow is illustrated in Figure 3.3. Compared with a Map-Side join, Reduce-Side joins are much more flexible as there are no restrictions placed on the structure or partitioning of the input data (White, 2010). However Miner and Shook (2012) state that the major disadvantage of this method is that all data required for the join will be passed through the Map/Reduce shuffle phase, thus incurring a costly network transfer stage.

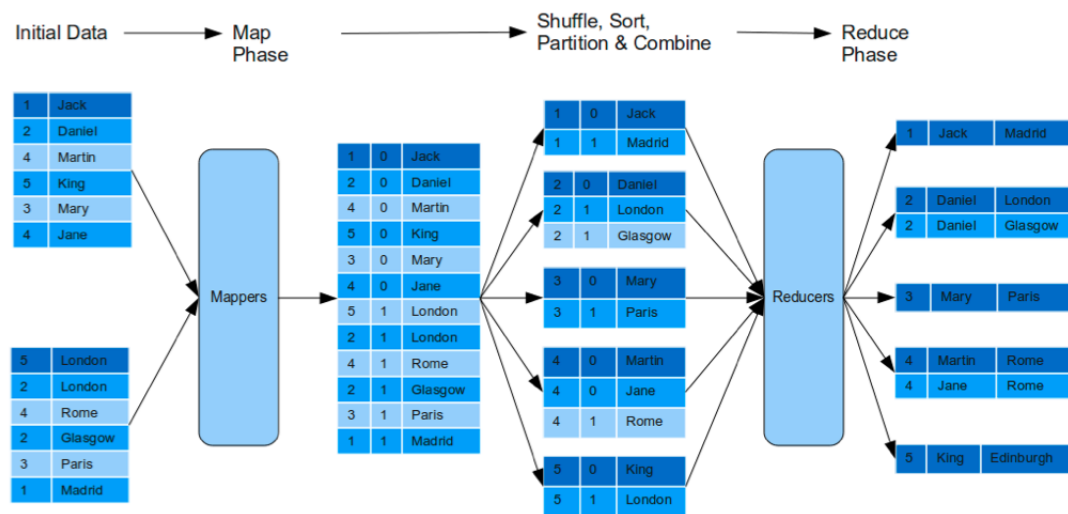


FIGURE 3.3: Reduce-Side Join (Chandar, 2010).

Afrati and Ullman (2010) show the logic behind the Reduce-Side to be relatively simple. For example joining the following datasets:

$$(P(a, b) \bowtie Q(b, c))$$

could be achieved by first iterating through both datasets P and Q in the Map function and emitting b as the key and a as the value for P , and b as the key and c as the value for dataset Q . The values would be grouped via the common key and the Reduce function would be presented with the key/value pair of - $b(a_n, c_n)$. The reduce function would then join the values together to create the final join result:

$$(P(a, b) \bowtie Q(b, c) \rightarrow FinalResult(a, b, c))$$

While this approach is suitable for joining two elements, joining multiple elements that depend upon the output of previous joins is not possible via this method. However using a technique called a Cascade join it is possible to process such situations (Chandar, 2010).

3.3.3 Cascade Join

The Cascade join is a method that allows multiple dependant relations to be joined via Map/Reduce. According to Afrati and Ullman (2010) a Cascade join is effectively a pre-determined set of Reduce-Side joins, processed in an iterative manner. Afrati and Ullman (2010) verify the logic behind the Cascade Join by demonstrating the joining of three datasets:

$$(R(a, b) \bowtie S(b, c) \bowtie T(c, d))$$

Using the first of two Reduce-Side joins, it is possible to join dataset R and S via the join key b :

$$(R(a, b) \bowtie S(b, c) \rightarrow IntermediateResult(a, b, c))$$

Then by employing a second Reduce-Side join it is possible to join the Intermediate result to dataset T via the join key c to create the final result.

$$(IntermediateResult(a, b, c) \bowtie T(c, d) \rightarrow FinalResult(a, b, c, d))$$

As the Cascade join is a set of Reduce-Side joins, it inherits many of the same advantages and disadvantages. But according to (Chandar, 2010) it has some additional benefits and drawbacks: A key advantage to Cascade joins is that the only limiting factor to the size and number of datasets that can be joined is the available HDFS space. However as each join is a complete Map/Reduce pass, the time overhead needed to setup each one will be incurred for each required pass. Cascade joins also inherit the network shuffle costs from the Reduce-Side join, but multiplied via the number of required passes. Another disadvantage is that the intermediate join results can consume vast amounts of space on the HDFS. However these intermediate results can be deleted once the final join result has been returned.

3.3.4 Broadcast Join

Blanas et al. (2010) state that if one of the datasets to be joined is of a small enough size, it could be stored in memory. This approach, entitled the Broadcast join, can be used as a performance optimisation for either a Map-Side or Reduce-Side join. A Map-Side join stands to have the largest performance gain from a Broadcast join, which comes from the lack of overhead from the Map/Reduce shuffle (Chandar, 2010). According to Lin and Dyer (2010) a Broadcast join functions by loading the smaller dataset in memory on all machines that will run a Map function. The Map function can then probe the dataset loaded in-memory to check for joins possible with the dataset passed to the Map function via the HDFS. Blanas et al. (2010) suggest the process can be optimised by storing the smaller dataset in an efficient data structure such as a Hash-Table, allowing for faster checking of element membership.

3.4 Feasibility Of Using Hadoop For RDF Processing

The need for a new and scaleable way to process the increasing amount of RDF data is apparent. Even some of the most distributed triplestores only use their parallelism to enable a larger volume of data to be stored, but do not increase the query performance (Patchigolla, 2011). This need for a new approach to parallel RDF data management has

led some researchers to consider the use of Hadoop to solve the problem. According to Soule (2011) Hadoop does not at first appear an ideal solution for processing RDF data, but if handled correctly has the potential to be a viable solution. Hadoop's intrinsic ability to scale means it could adapt to fit as RDF data-sets continue to increase in size. Myung, Yeon, and Lee (2010) further argue that RDF's simple data model along with potential for RDF to be rendered as text strings via the N-Triples notation, serve to enhance the case for investigating Hadoop as an RDF processing platform.

3.5 Strategies For Storing RDF Data On The HDFS

One of the first points to consider when designing any Hadoop application is how the input Data will be stored on the HDFS. As discussed in section 2.1.3, RDF has the potential to be serialised in a variety of different formats. Husain (2009) has investigated the storage of RDF data and argues that from the available formats, the least suited would be RDF/XML as it requires several lines to represent a single statement. He argues that as Hadoop processes each line of input data individually, a much more suited RDF format would be N-Triple, as it allows a complete RDF statement to be rendered in a single line of text. The ability of Hadoop to process a complete triple on each iteration has clear performance benefits, as individual triples can be extracted without the need to parse the entire input (Husain et al., 2009).

Husain et al. (2009) advocate splitting the input data on the HDFS into smaller chunks before allowing it to be processed. They propose a system whereby the original RDF input data is split into smaller and more manageable chunks. Firstly the data is divided based on the RDF Predicate, this stage is called the Predicate Split. Work conducted by Stocker, Seaborne, Bernstein, Kiefer, and Reynolds (2008) shows that many real-world RDF datasets do not contain more than twenty different predicates. Splitting the original input data into even twenty smaller chunks would enable any query to take a much more relevant selection of input data, thus decreasing the amount of time spent seeking data requested via the query (Husain, 2009). Husain (2009) further suggest splitting the data again once the Predicate

Split is complete, but this time the split, being based on the RDF Object, could further target more relevant data to a query.

Rohloff and Schantz (2010) store the RDF data as raw plain text N-Triple files, housed directly on the HDFS. However rather than storing each triple as a unique entity, a pre-processing stage is introduced, to store the RDF in a non-native representation. This representation stores every triple for a certain subject on the same line of text. Rohloff and Schantz (2010) demonstrate how this representation works by showing how their system would store three triples with the common subject of Pub1:

```
Pub1 :author Prof0 :name "Pub1" a :Publication
```

As Pub1 only needs to be printed once, this representation also acts as a rudimentary form of data compression, cutting down on the amount of HDFS space being used.

Goasdoué et al. (2013) store the RDF data in a novel way, exploiting the HDFS data replication to enable faster query processing. As highlighted in section 2.5, by default the HDFS will replicate each block of data three times, with each replicated block being distributed to a unique node wherever possible. In the approach presented by Goasdoué et al. (2013), each block of RDF data is still replicated three times, but instead of each replication being identical, one is partitioned on the subject of a triple, one on the property, and the last on the object. Further to this partitioning, the system stores any identical permutations of a certain RDF subject, predicate or object on the same HDFS data node.

3.6 Queries on RDF Data Using Map/Reduce

3.6.1 Existing Theoretical Approaches

A number of different Map/Reduce-based approaches to query RDF data stored on a HDFS have been explored by various researchers. Many of the approaches are designed to provide a complete SPARQL query engine that translates a given query into a series of Map/Reduce jobs with little or no input from the user.

Rohloff and Schantz (2010) introduce a system entitled SHARD, which was one of the first coherent efforts to query RDF data using Hadoop. They designed the system to complete two goals: to serve as a persistent storage for RDF data and to provide a SPARQL endpoint to query the stored data. As discussed in section 3.5 they employ a method of storing the RDF data on the HDFS so that every triple with a common subject is rendered on a single line of input data. According to Rohloff and Schantz (2010) to complete a response to any given query, the SHARD system spawns a series of Map/Reduce iterations. The first iteration maps all the input triples to a list of variable bindings denoted by the first clause in the SPARQL query. Any triples which match the clause are passed to the reduce and then saved to the HDFS. This step is repeated n number of times for each of the clauses in the query, with the input being the original data plus the output from all previous stages. Using this method allows the system to filter out any previously selected triples which do not meet the new selection criteria. These intermediate stages join the newly selected triples to the previous results via a reduce side join. This allows the system to pass only those triples which meet all the current clauses onto the next iteration. According to Rohloff and Schantz (2010) once all the clauses have been completed, the final Map/Reduce stage is used to complete the SELECT element of the query by making sure only the variables originally requested are presented to the user via the final output.

Kulkarni (2010) presents a system which expands on the Jena ARQ query engine and attempts to transfer the processing of data to Hadoop. The approach is split into two distinct phases, with each phase implemented as at least one complete Map/Reduce iteration. Firstly a selection phase is run to select all the triples required to complete the query. The selection phase groups the final output based on the order of the patterns from the BGP. Figure 3.4 shows how, with a query comprising two BGP elements, the selection phase splits the reducer output into two separate files. The output is formatted with the key being equal to the pattern number and the complete triple as the value.

Kulkarni (2010) further explains that following the selection phase, the relevant triples are passed to the join phase, in which any required joins are performed in an iterative fashion. In this phase any common elements from the previously generated pattern files are joined together to complete the query. Figure 3.5 shows how the join phase mapper selects joining

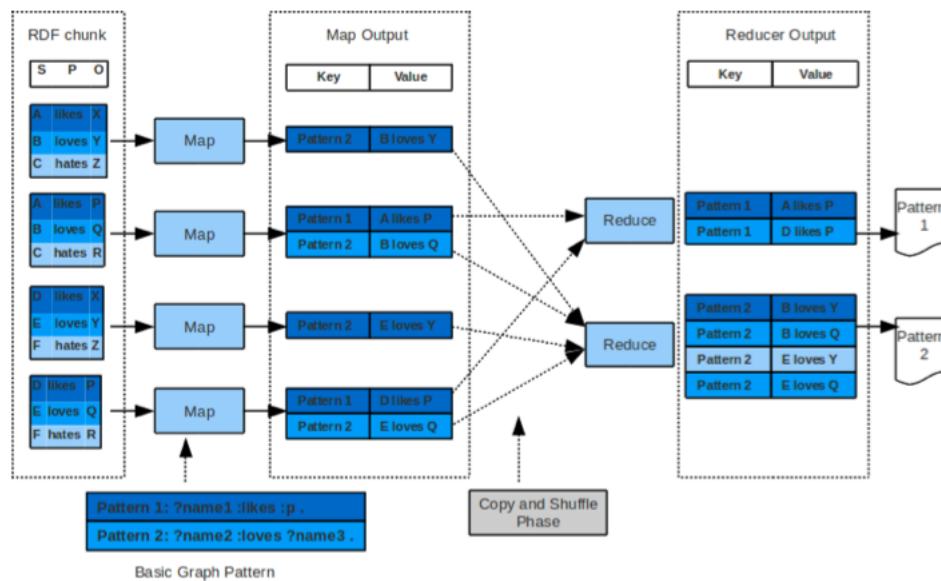


FIGURE 3.4: Kulkarni's Selection Phase (Kulkarni, 2010).

elements from the triples, setting the element to be joined as the key and the full triple as the value. Leveraging the shuffle and sort phase highlighted in section 2.5, will ensure that any common elements will be sent to the same reducer and thus can be joined together to create the final output. Joining via this method means that a large number of triples could be joined in a single pass, providing they share a common element.

Mazumdar (2011) presents a similar system with the RDF data being first processed by a selection phase and then a join phase. However this implementation introduces an additional stage entitled the projection stage. While each stage is similar in role to Kulkarni's work, there are slight variations in operation. Mazumdar (2011) states that his selection stage is used to filter out those triples not required by the query. This stage's output is a selection of triples which match at least one of the query patterns, split into separate files on the HDFS based on the pattern number. This is then passed to the next Map/Reduce task which performs a Reduce-side join, running iterative passes of the same job if multiple joins are required. After the Join phase, the data is passed through a concluding stage, which projects the requested final output from the original query. Processing RDF data using this approach will require at least three different Map/Reduce iterations to complete any given query, expanding by one additional iteration per extra join required (Mazumdar, 2011).

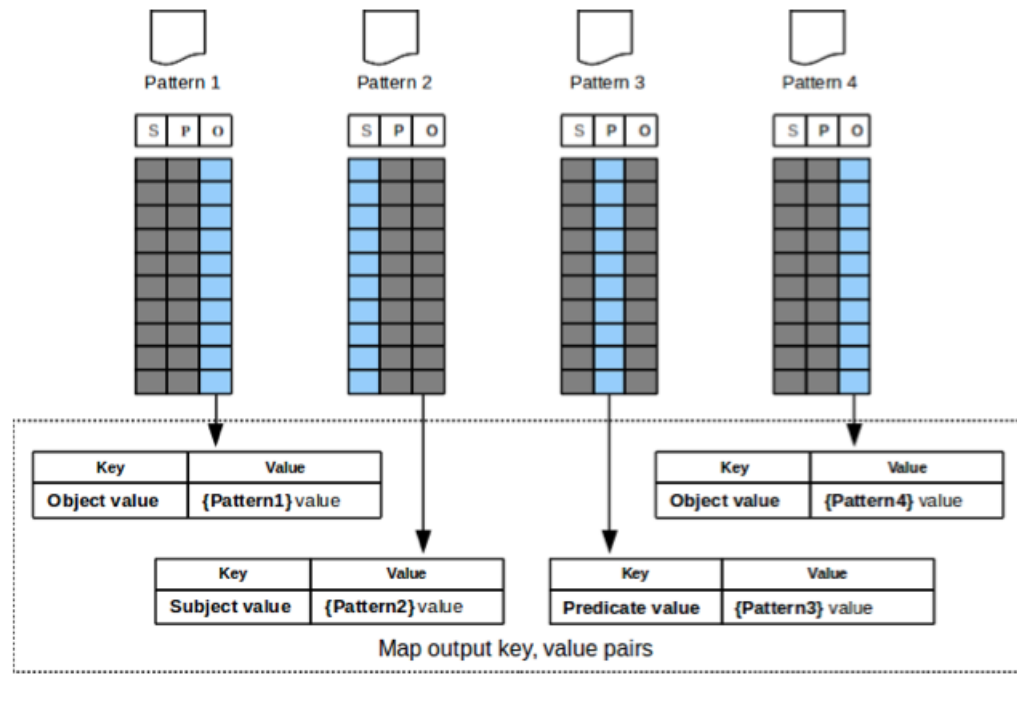


FIGURE 3.5: Kulkarni's Join Phase (Kulkarni, 2010).

Mazumdar (2011) demonstrates how the join phase would iteratively process a certain SPARQL query shown in Figure 3.6. Figure 3.6 also shows how each of the different triple patterns are numbered for use later in the query planning stage.

```

SELECT ?name1 ?name2
WHERE {
  ?article1 rdf:type bench:Article . //Numbered as 0
  ?article2 rdf:type bench:Article . //Numbered as 1
  ?article1 dc:creator ?author1 .    //Numbered as 2
  ?author1 foaf:name ?name1 .        //Numbered as 3
  ?article2 dc:creator ?author2 .    //Numbered as 4
  ?author2 foaf:name ?name2 .        //Numbered as 5
  ?article1 swrc:journal ?journal .  //Numbered as 6
  ?article2 swrc:journal ?journal .  //Numbered as 7
}

```

FIGURE 3.6: Mazumdar's Example SPARQL Query (Mazumdar, 2011).

Figure 3.7 shows the query plan, represented as an RDF graph, that the rest of the join phase would follow. Mazumdar (2011) states that the query plan is generated based on triple patterns that share a common variable on which they will be joined. For example pattern 0,2

and 6 share the common variable ?article1, so they would be joined in the first Map/Reduce join pass.

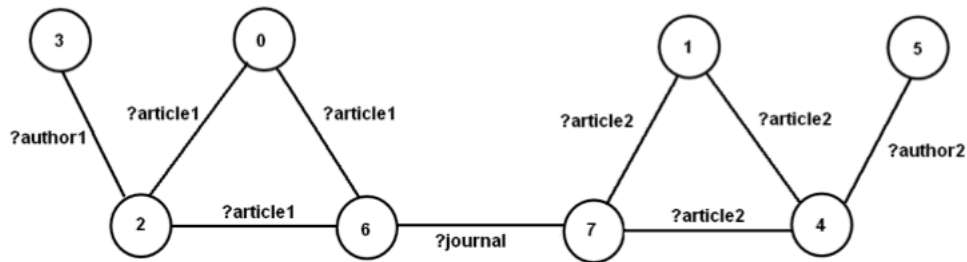


FIGURE 3.7: Mazumdar's Query Execution Plan (Mazumdar, 2011).

The resulting output, with the completed join of patterns 0,2 and 6 highlighted in blue, is shown in Figure 3.8. The join phase would continue in this manner, joining all elements sharing a common variable via a single Map/Reduce task. According to Mazumdar (2011), using this method would take a further four iterations to produce the final join stage output. This would then serve as the input to the projection stage which performs the 'SELECT' clause of the query shown in Figure 3.6, in this case outputting the request variables of ?name1 and ?name2.

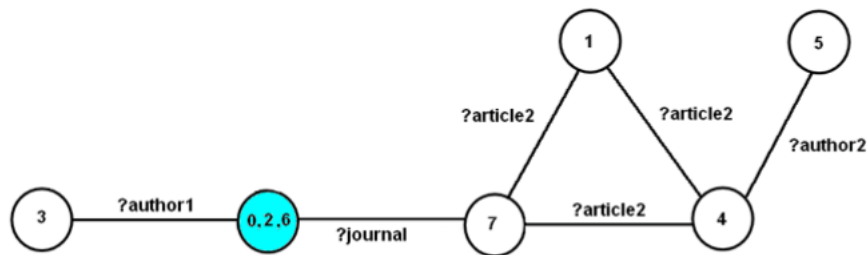


FIGURE 3.8: Mazumdar's First Join Iteration (Mazumdar, 2011).

Husain, McGlothlin, Masud, Khan, and Thuraisingham (2011) have created a framework called HadoopRDF. It is highlighted in work conducted by Goasdoué et al. (2013) as being one of the most advanced of the current implementations created to query RDF data via Map/Reduce. Similar to the method presented by Husain et al. (2009), the HadoopRDF system splits all the input triples into separate files based on the predicate. Triples with the predicate of `rdf:type`, which they highlight as being the most common predicate, are again further split into separate files based on the object. To evaluate any given query firstly the

system evaluates which of the pre-split files are required as input for the job. Then, Husain et al. (2011) explain that the system implements an heuristic approach to find the optimum method to complete the query. This heuristic approach to finding the best query plan is what differentiates HadoopRDF from other implementations and, due to its ability to cut the number of Map/Reduce iterations, is a key driver of its performance (Goasdoué et al., 2013). The heuristics-based approach uses a greedy algorithm to determine the minimum numbers of jobs in which a set query can be completed. Once the appropriate plan has been generated, the system then runs the required number of jobs in a pre-determined order. The input for a certain job is the output from the previous job, with the original input being the required selection of split RDF files. If any joins are required on different variables, HadoopRDF processes them using iterative reduce-side joins. However, HadoopRDF will join any number of identical variables in a single pass in an effort to reduce the number of required passes (Husain et al., 2011).

Goasdoué et al. (2013) present an alternative Map/Reduce based RDF query system which borrows the idea of cliques from graph theory to perform certain queries in a highly efficient manner. They propose a system of two different clique-based query classifications. Firstly, they propose the 1-clique query, which are only those queries which join triples via a single common variable. Figure 3.9(a) shows a SPARQL query which, based on the single common join variable `?x`, would be classified as a 1-clique query. Figure 3.9(b) shows the graph representation of the same query, with each node being a single triple pattern, connected by their common join variable of `?x`. To implement the 1-clique query via Map/Reduce Goasdoué et al. (2013) exploit their data replication strategy discussed in section 3.5. The strategy ensures that any triples which share at least one element are accessible on a single node. They are then able to exploit the ability of map side joins, highlighted in section 3.3.1, to perform joins on a single common key in a map only job, bypassing the costly shuffle/sort phase. Goasdoué et al. (2013) performed an analysis of real-world SPARQL queries taken from DBPedia before they designed the system. They observe that nearly 99% of queries from the real-world sample have the potential to be processed via a 1-clique query.

Secondly, Goasdoué et al. (2013) propose the central clique query, defined as a series of patterns connected via one overlapping element. Figure 3.10(a) shows a SPARQL query

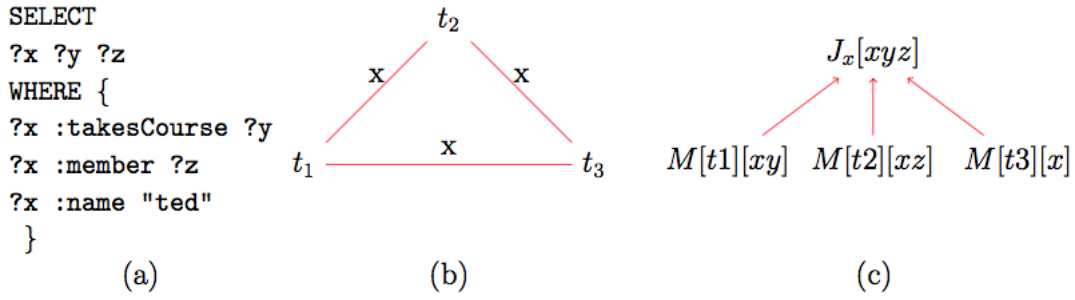


FIGURE 3.9: Goasdoue 1-Clique Query (Goasdoué et al., 2013).

which would qualify as a central clique query. Figure 3.10(b) shows the same query represented in graph form, illustrating how the two different collections of triple patterns are connected via a single element. To implement a central clique query in map/reduce, Goasdoué et al. (2013) exploit a map side join and then a reduce side join (discussed in section 3.3.2), to enable two different triple groups to be joined. For example the query from Figure 3.10 could be completed in one Map/Reduce iteration. Firstly as the query contains two distinct 1-clique queries, each of these would be completed in the map phase as before. Then the required additional join on the two separate patterns would be accomplished in the reduce stage by setting the common variable as the intermediate key. Using this method, any number of different interconnected 1-clique patterns could be joined by using iterative Map/Reduce passes.

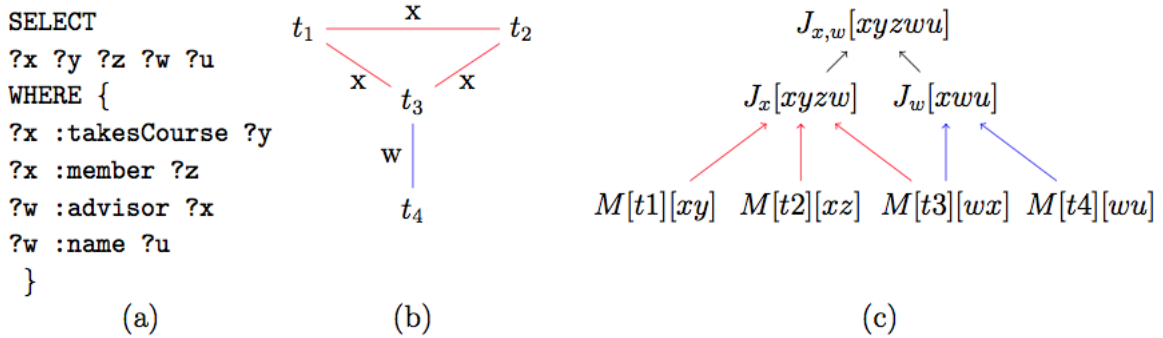


FIGURE 3.10: Goasdoue's Central Clique Query (Goasdoué et al., 2013).

3.6.2 Presented Performance Results

This section will explore and compare relevant practical results presented in the literature detailed in the previous section.

Mazumdar (2011) presents results from the developed solution on a 20 node cluster. To evaluate the performance, RDF data and SPARQL queries from the SP2 Benchmark were used. The SP2 Benchmark is a commonly used tool to assess the performance of triple-stores. It comprises a set of relatively simple SPARQL queries which are run across a variety of dataset sizes. Full details of SP2 are given by Schmidt, Hornung, Lausen, and Pinkel (2008). Figure 3.11 shows the results for the solution (Mazumdar, 2011). The results show that for even relatively simple queries running on a dataset size of 50 million triples, the solution takes up to 50 minutes to return a response.

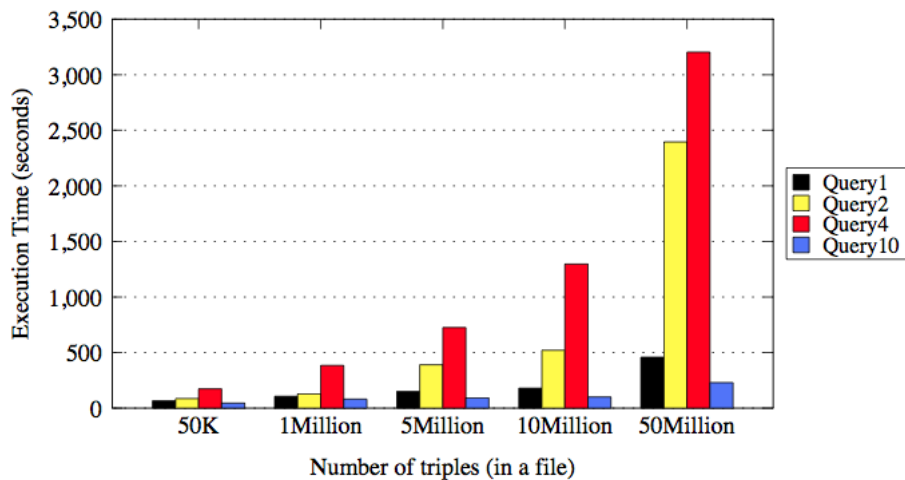


FIGURE 3.11: Mazumdar's Performance Results (Mazumdar, 2011).

Husain et al. (2011) assess the performance of their solution called HadoopRDF by running it on a 10 node cluster. They test HadoopRDF at uploading and querying RDF data from the LUBM benchmark, against Jena running on a single machine and in main memory. Full details of LUBM are given by Guo, Pan, and Heflin (2005). The LUBM benchmark comprises an RDF data generator and a series of SPARQL queries. For the test, they performed a range of LUBM SPARQL queries on RDF data sets comprising different numbers of triples. Results from the solution are shown in Figure 3.12. The tests show that for a smaller volume of triples, Jena is faster. However the HadoopRDF system starts to become faster once the

amount of triples is increased past 25 Million. They also show that Jena is unable to even process datasets of more than 100M triples. The results also show that the solution is able to perform a single SPARQL query on one billion triples in under one hour. Husain et al. (2011) also test the scalability of HadoopRDF by testing how the framework performs on datasets of up to 6.6 billion triples. They find that the system features a sub-linear increase in query time against dataset size, meaning that an increase in dataset size does not result in a proportional increase in the time taken to query it.

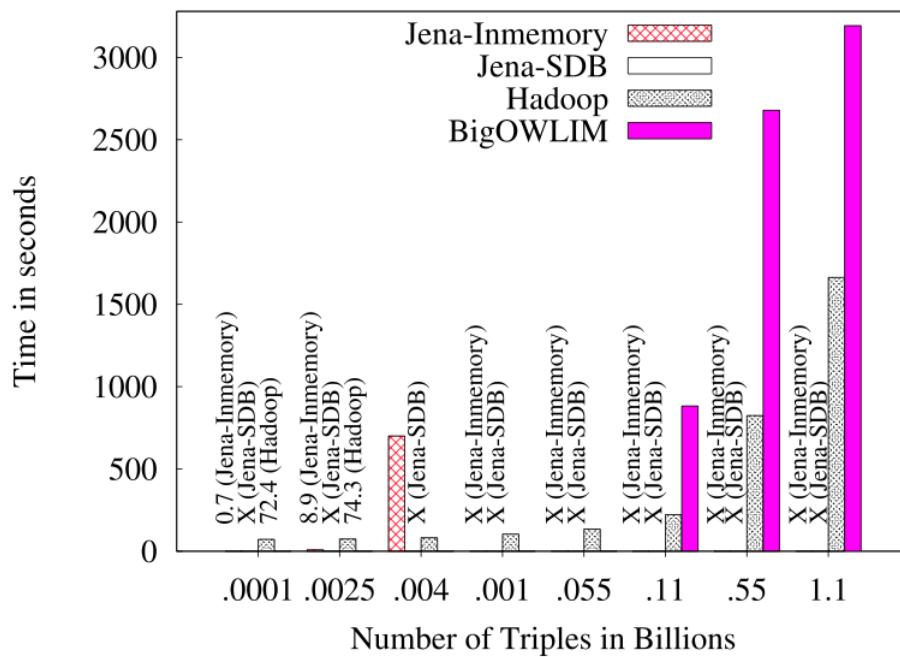


FIGURE 3.12: Husain's Performance Results (Husain et al., 2011).

Goasdoué et al. (2013) present results from their solution which is tested on an eight node cluster. As well as assessing their own work, they also compare it to the HadoopRDF system created by Husain et al. (2011). They are also using the LUBM Benchmark to test the upload and query performance of the system. Figure 3.13 shows the upload performance of the system on dataset sizes of one and two billion triples. The results show that the system required 257 minutes to upload one billion triples and the system failed to upload two billion triples over the eight node cluster.

Figure 3.14 shows the query performance of the system when running on a dataset size of one billion triples. The results show that the system is much more effective than HadoopRDF across the range of LUBM SPARQL queries, with the most complex query being completed

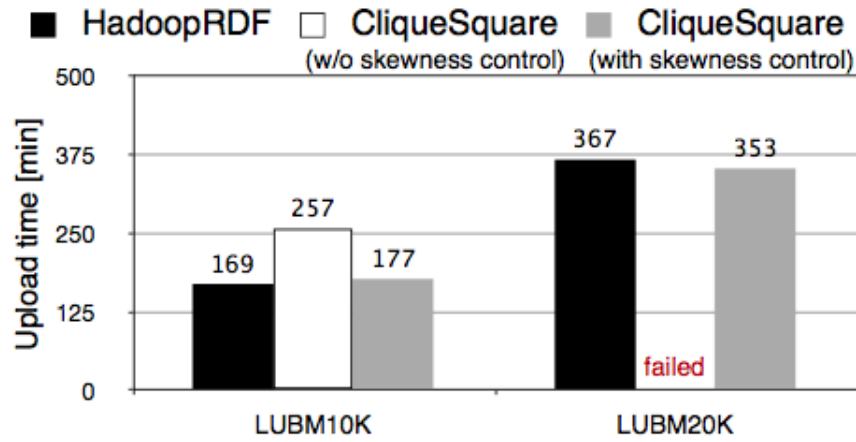


FIGURE 3.13: Goasdoue's Upload Performance Results(Goasdou  et al., 2013).

in 59 minutes. These results show that the solution presented by Goasdou  et al. (2013) has the best performing query stage of the current SPARQL over Hadoop processing solutions as detailed in section 3.6.

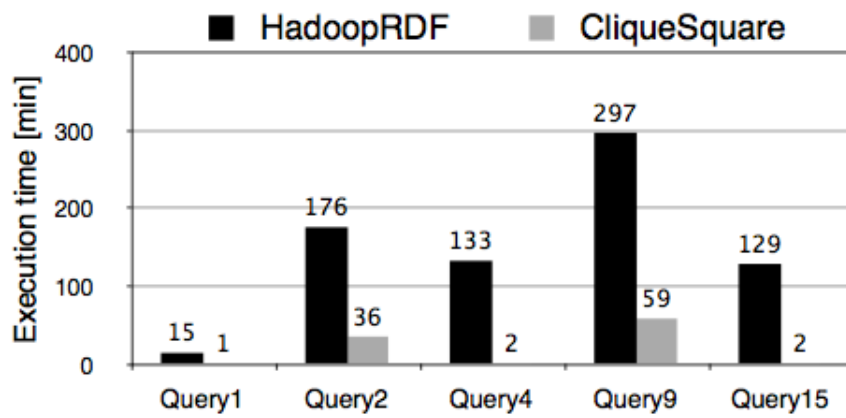


FIGURE 3.14: Goasdoue's Query Performance Results(Goasdou  et al., 2013).

3.7 Limitations Of Existing Hadoop RDF Solutions

The systems introduced in the literature from the previous sections are all designed to process generic SPARQL queries and are therefore not optimised to any specific case. All the current implementations only present details of how the systems cope with relatively simple

SPARQL queries from the LUBM or SP2 benchmarks. None give details on how the systems cope when required to perform large numbers of joins which a complex SPARQL query would entail. In addition, none of the current solutions explore the possibility of grouping common joins from different unique SPARQL queries together to save on re-computation.

The following two sections explore two of the main limitations that the current systems share.

3.7.1 Hadoop Joining Strategies

Many of the existing projects which tackle processing RDF over Hadoop rely on the slower reduce side join, meaning that they would require many iterations to process complex queries. Few attempts have looked at the possibility of a map-side or broadcast join to create highly optimised systems. The work presented by Mazumdar (2011), Rohloff and Schantz (2010), Husain et al. (2011) and Kulkarni (2010) all perform any required joins via a series of cascade reduce-side joins. As described in section 3.3.3, cascade joins involve a lot of data transfer over the network as well as incurring multiple JVM initialisation stages, both of which will lead to poor performance. From the currently available literature, only one solution has been developed which attempts to use a map-side join when processing SPARQL queries. This is the clique-based approach presented by Goasdoué et al. (2013). This approach uses a map-side method to join triples which share a common element but falls back to a reduce side method to join additional elements. There appear to be no available solutions which utilise the highly efficient broadcast join method.

A selection of the current solutions, for example work presented by Kulkarni (2010) and Rohloff and Schantz (2010), will run one complete Map/Reduce iteration per line of SPARQL query. This approach ignores the multi-way join method which groups the joining of any common elements into a single Map/Reduce reduce-side join iteration. As many SPARQL queries contain common join variables, a system which is unable to exploit this to improve performance does not appear highly optimised.

3.7.2 Data Upload Stages

Many of the current solutions rely on pre-processing of the data to enable faster querying at a later date. However these data upload stages often change the structure of the data, thus removing the characteristic triple format and element order of the RDF data to be stored. For example the solution presented by Husain et al. (2009) split the original RDF dataset into different files based on the predicate, meaning that the files would have to be reconstituted if the user wanted to extract the complete original RDF dataset. In the example presented by Goasdoué et al. (2013), the order of the RDF elements is altered for each of the three data replications, meaning that the user would have to re-convert the data back to its original order of subject, predicate and object. The data upload stage presented by Goasdoué et al. (2013) also has a further limitation, in that it can only be used to join dataset sizes less than the size of a HDFS block; by default this is 64MB.

These approaches would need additional Map/Reduce jobs to convert the data back into the original RDF format. None of the currently available literature explores the possibility of storing and querying RDF data over Hadoop in its native format. There may very well be good performance-related reasons for this, but as all the upload stages presented in the literature take considerable time, the possibility of not requiring one should be explored.

3.8 Possibility Of A Highly Optimised Solution

From reviewing the current literature it is possible to see that there are gaps which would enable a highly optimised system that stores and Queries RDF data using Hadoop to be created. Firstly, the case can be made for exploring a system where prior knowledge of the data to be stored informs a highly optimised system. Knowledge of the structure and distribution of any RDF data could be utilised to enable efficient map-side and broadcast joins to be used. For example, triples which contain common elements could be grouped together to enable map-side joins. Also smaller groups of triples which are required to be joined to larger groups could be made available via a broadcast join. This would save on numerous costly iterative reduce-side joins which the current solutions rely upon.

Secondly, the case can be made for designing a system in which knowledge of the type of SPARQL queries to be performed allows for optimised query planning. Currently none of the existing system use knowledge of the SPARQL queries which will be performed to save on costly re-computation. This means that the current solutions would recompute all the required joins for two SPARQL queries in which only one join was different. A highly optimised solution would not waste time and resources re-performing joins, instead it would only perform the additionally required join. This could be achieved by creating a super query from an input of multiple queries, so that common joins would not be recomputed. In addition, the intermediate join data could be stored on the HDFS so any future queries which require the same joins could utilise it.

Chapter 4

Analysis Of Current Jena TDB Implementation

4.1 Current Implementation

This chapter will introduce and analyse the pre-existing approach used to assess the quality of medical data using linked data technologies. This approach was developed by Dr David Corsar and Dr Laura Moss and full details are given by Corsar et al. (2012) and Moss, Corsar, and Piper (2012).

Corsar et al. (2012) state that this framework can be broken down into three key stages: firstly, pre-existing medical data is converted into RDF data and stored in Jena TDB. Secondly, the data can be annotated with provenance information, such as the specification of the machines which recorded the data. Lastly, a data-checking component assesses the quality via a series of SPARQL rules. Figure 4.1 shows the complete framework. The most important of these steps is the data checking stage, which according to Corsar et al. (2012), comprises a series of SPARQL queries which test various qualities of the data, including checks on acceptable data ranges and missing data points. If a potential error is found the system will annotate it with one of two different states based on the level of confidence that it is truly an error. The two states are 'Possible Error' and 'Probable Error' (Corsar et al.,

2012). The function of the data checking stage is to highlight any potential errors in the dataset to the user before the data is utilised for other purposes.

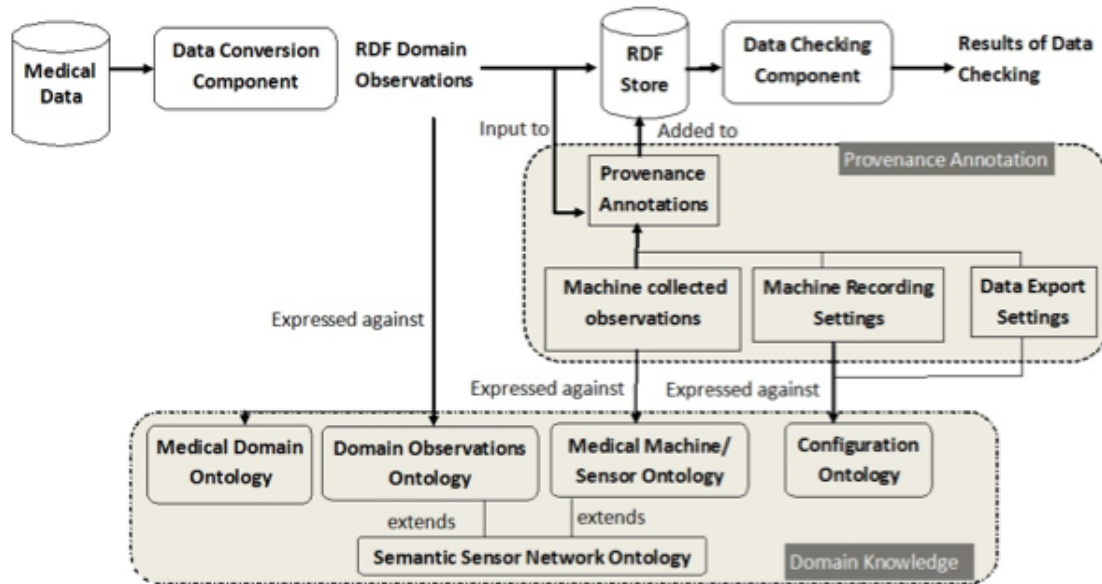


FIGURE 4.1: Current Approach For Checking Medical Data (Corsar et al., 2012).

4.2 Limitations of Existing Solution

While the current framework functions correctly in the detection of errors, it suffers from two main performance issues, details of which are given in the papers describing the framework. Firstly Corsar et al. (2012) explain that their original method of using Jena TDB took over six hours to upload 1.6 Million triples. Secondly, Moss et al. (2012) explain how the framework, when working on an RDF dataset comprising 609,168 triples, took over 150 minutes to return a response for a single SPARQL query. Both these performance issues are related to the same Semantic Web component of the framework; Jena TDB.

The performance of the current framework will be tested in the following section.

4.3 Current Implementation Performance Analysis

To assess the real-world performance of the Jena-based framework, it was benchmarked against a range of RDF dataset sizes. Hardware and software specifications of the machine on which the benchmarks were run can be seen in section 6.1.1. Both the upload and query time for the Jena implementation were benchmarked.

A more detailed and comparative analysis of the Jena framework can be found in section 7.5.

4.3.1 Upload Time

Figure 4.2 shows the upload performance of Jena TDB across a range of data set sizes. As the results show, the upload performance of Jena TDB is poor once the dataset size increases past 32 million triples, as Jena took 157 minutes to upload 64 million triples. While this upload performance is not as poor as the claims made by Corsar et al. (2012) and Moss et al. (2012), direct comparisons between the two results is impossible due to the lack of knowledge about the test environment and datasets used by Corsar et al. (2012) and Moss et al. (2012). However both results due suggest that Jena is not able to cope with storage demands made by the massive potential volumes of NHS RDF data.

4.3.2 Query Time

To assess the query performance of Jena, all of the required queries were run upon the data sequentially. The results from the queries were written to disk. Figure 4.3 displays the results from the test. It shows that as the number of triples increases, so does the time taken to return query responses. As with the upload test, once the number of triples increases past 32 million, the query performance begins to suffer, with Jena taking over one hour to complete all the queries (Found in appendix A) on a dataset size of 64 million.

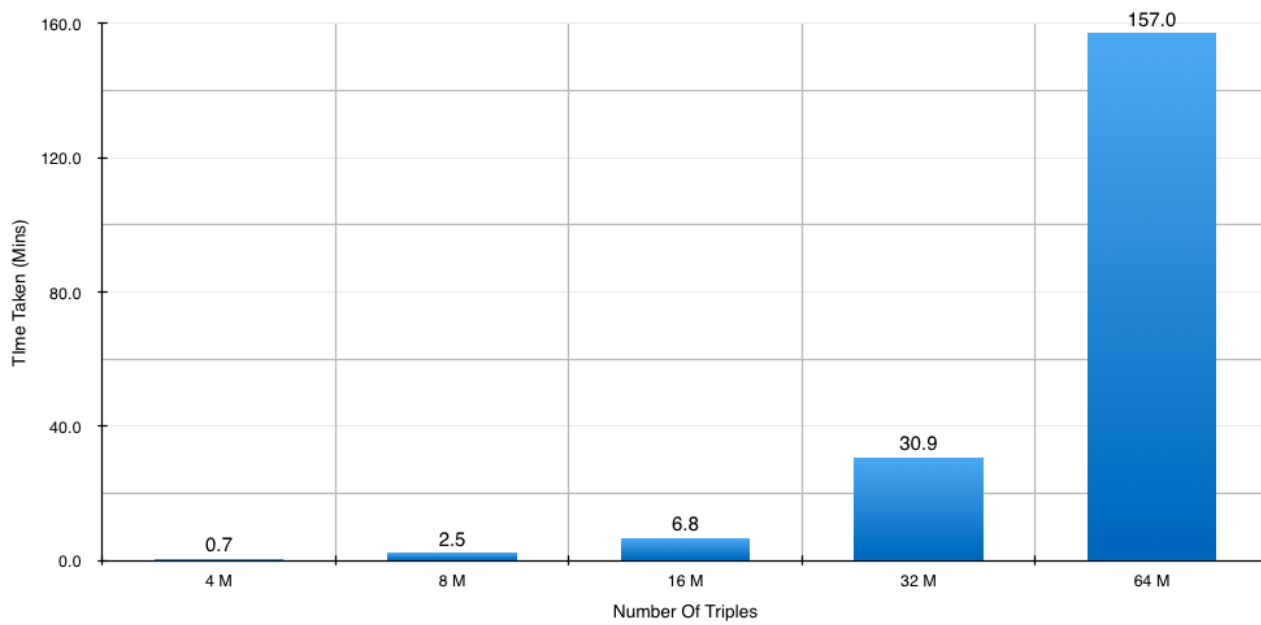


FIGURE 4.2: Jena TDB Upload Performance

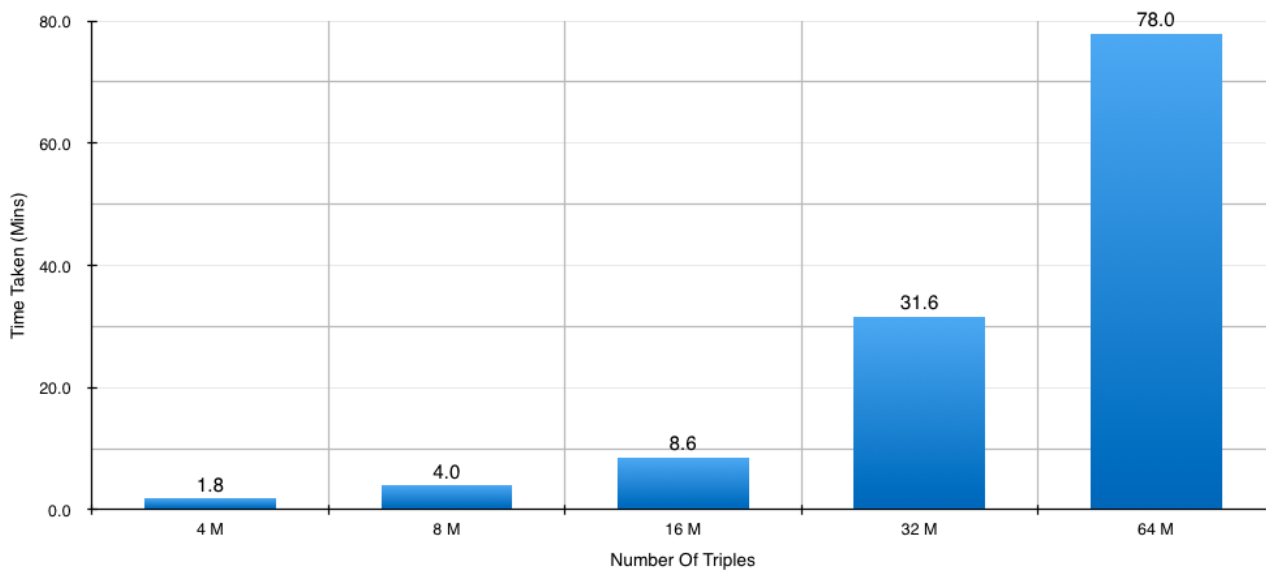


FIGURE 4.3: Jena TDB Query Performance

4.3.3 Analysis

As the results from the previous section shows, the current Jena TDB and SPARQL implementation suffers from performance limitation, even when running on a modern desktop

machine. This machine is detailed in section 6.1.1. The results for the upload and query stages produce unsatisfactory results when the number of triples is increased to 64M. Indeed Jena was not able to return a result when the dataset size was increased to 128 million triples. Neither the upload nor query performance is as poor as (Corsar et al., 2012) and (Moss et al., 2012) found, however this could be explained by a performance disparity between the underlying machines used in the test. .

4.4 The Need For A New Framework

The results from this test suggest that Jena is not the platform most suited to perform error checking on NHS scale RDF datasets. Aside from the performance issues, Jena is only able to upload to and query data from a single machine, meaning that there will always be a limit, determined by the size of HDD, on the volume of RDF data that could be processed. These results show that a new way of assessing the quality of medical data needs to be developed to be able to cope with the potential volume of RDF NHS data. Logically, the new framework would scale the work and storage load across multiple machines to enable more data to be stored and queried. According to Asanovic et al. (2009), this move to a multi-core and multi-machine approach is being experienced across the whole spectrum of the computer-science field.

Chapter 5

Hadoop Implementation and Algorithm Design

As shown in Chapter 4, there is a clear need to produce a new framework that can cope with the potential volume of NHS RDF data. This chapter will detail how this new framework was first designed and then implemented.

5.1 Structure and Distribution Of The Real-World Medical Data

For any new framework to be designed, real-world medical data was required. Due to ethical and privacy reasons it was not possible to have access to large volumes of real medical data. However, three anonymised datasets were provided by the University Of Glasgow. The datasets all contain neurological data from the BrainIT project (Chambers et al., 2009). The datasets contain information regarding neurological readings taken from patients, along with information regarding the machines and sensors which produced these values. The provided datasets represent a small period of time for three different patients and combined they contain 7,933,649 RDF triples. More medical data was synthetically generated for use in later tests, details of this process can be found in section 5.2.

The rest of this section will explore the structure and distribution of the three real-world medical data sets.

5.1.1 Subject, Predicate and Object Distribution

To better understand the distribution of the data, the number of unique RDF Subjects, Predicates and Objects were obtained using a simple Map/Reduce job. The results of which can be seen in table 5.1.

RDF Element	Number Of Elements
Subject	1,523,106
Predicate	61
Object	1,711,702

TABLE 5.1: Subject, Predicate and Object Distribution

The results show that only 61 unique predicates are used throughout all of the datasets. Further, it can be seen that the number of unique subjects and objects are closely matched. It also shows that compared to the total number of triples (7,933,649), many of the subjects and objects are replicated numerous times.

5.1.2 Distinct Triple Group Distribution

As will be explained in section 5.3, all of the original SPARQL queries join distinct groups of triples, which are linked by a common theme. Broadly, the distinct groups of triples are related to two things: patient recording values and permitted value ranges. For any possible optimisations for later queries, knowledge of the distribution of these triple groups would be required. In order to explore the distribution of these groups a Map/Reduce algorithm was devised that assessed the number of members in set triple groups. This was then run against the same three medical RDF datasets, grouping the triples into one of three groups. The groups were ?range, ?obs and ?cs and were chosen to reflect the triple groups used in the SPARQL queries. The result is shown in table 5.2.

Triple Group Name	Number Of Elements
?obs	3,426,188
?range	518
?cs	446

TABLE 5.2: Triple Group Distribution

The results show that the distribution of the triple groups is massively skewed towards the ?obs group. The ?obs group has 3,426,188 unique triple instances, which when compared with the total number of triples in the dataset (7,933,649) shows the ?obs group comprises a large proportion of the total triples. Both the ?range and ?cs groups contain very few instances.

5.2 Data Generation

Due to not having access to large volumes of medical data, it became apparent that data would have to be synthetically generated. As the framework needed to be tested against massive datasets, one billion triples were generated. To produce the new data, a Map/Reduce algorithm was developed which generated new RDF data based upon an input of real world medical RDF data. The algorithm retains the structure and distribution from the real world data, but inserts new randomly generated values for the variable triple elements. The algorithm exploits knowledge of the data so that it does not alter triples which are constant across all datasets, for example the from the ?range triple group.

To ascertain that the newly generated data matched the structure, distribution and triple group distribution of the real world, a generated dataset was run through the same algorithms used to assess distribution from section 5.1. This was done to ensure that a real-world and generated dataset of equal size contained the same number of unique subjects, predicates and objects, as well as an identical triple group distribution. This ensures that the SPARQL queries still return correct responses when run against the newly generated datasets.

5.3 Analysis Of Current SPARQL Queries

The current implementation comprises eight SPARQL queries which test a range of metrics within the data. In the current framework, these SPARQL queries are run sequentially and the required joins re-computed for each query.

5.3.1 Min/Max Queries

An example of two of the queries is shown below. These queries test the data for values outside of the minimum and maximum permitted ranges. Each of the queries comprises two main sets of BGPs: the ?range pattern which contains information about the expected range for the values and the ?obs pattern which contains the patient information and reading values.

```

---query.aboveMaxAcceptable
// check if a value is above the maximum acceptable value
SELECT ?obs ?p ?htime ?max ?value WHERE {
?range a med:AcceptableRange.
?range med:clinicalRangeMax ?max.
?range pd:hasParameter ?p.

?obs a mo:PhysiologicalObservation.
?obs ssn:observedProperty ?p.
?obs ssn:observationResultTime ?time.
?obs pd:atHumanTime ?htime.
?obs ssn:observationResult ?a1.
?a1 ssn:hasValue ?a2.
?a2 pd:readingValue ?value.
FILTER (?value > ?max)
}

---query.belowMinAcceptable
// check if a value is below the minimum acceptable value
SELECT ?obs ?p ?htime ?max ?value WHERE{
?range a med:AcceptableRange.
?range med:clinicalRangeMin ?min.
?range pd:hasParameter ?p.

?obs a mo:PhysiologicalObservation.
?obs ssn:observedProperty ?p.
?obs ssn:observationResultTime ?time.
?obs pd:atHumanTime ?htime.
?obs ssn:observationResult ?a1.
?a1 ssn:hasValue ?a2.
?a2 pd:readingValue ?value.
FILTER (?value < ?min)
}

```

Looking at the two queries, it is apparent that they are extracting nearly identical triples from the datasets; the only difference being the extraction of either the min or max from the range group.

5.3.2 More Complex Queries

While the min/max queries are relatively complex, requiring the joining of around 12 unique triples in order for successful completion, the current implementation features more complex rules. These rules check the given value against the accuracy of the sensor by which the reading value was recorded and also for medical conditions that may be affecting the values. Two examples of the more complex rules are shown below. The first rule shown below checks if a given value is above minimum acceptable value, but when sensor accuracy is considered, is actually below the minimum. The second rule checks if value is above the maximum acceptable value which may be explained by the medical condition Hypertension.

```
---query.aboveMinAcceptableMinusSensorAccuracy2
```

```
SELECT ?p ?min ?value ?htime WHERE{
?range a med:AcceptableRange.
?range med:clinicalRangeMin ?min.
?range pd:hasParameter ?p.
```

```
?obs a mo:PhysiologicalObservation.
?obs ssn:observedProperty ?p.
?obs ssn:observationResultTime ?time.
?obs pd:atHumanTime ?htime.
?obs ssn:observedBy ?sensor.
?obs ssn:observationResult ?a1.
?a1 ssn:hasValue ?a2.
?a2 pd:readingValue ?value.
```

```
?sensor ssn:hasMeasurementCapability ?mc.
```

```
?mc a ssn:Accuracy.
?mc ms:capabilityValue ?accuracy.
FILTER (?value > ?min)
LET (?v2 := ?value * ?accuracy)
FILTER ((?value - ?v2) < ?min)
}
```

```
---query.aboveMaxAcceptableOkHypertension
```

```
SELECT ?obs ?p ?htime ?max ?value WHERE {
?range a med:AcceptableRange.
?range med:clinicalRangeMax ?max.
?range pd:hasParameter ?p.
```

```
?obs a mo:PhysiologicalObservation.
?obs ssn:observedProperty ?p.
?obs ssn:observationResultTime ?time.
?obs pd:atHumanTime ?htime.
```

```
?obs ssn:observationResult ?a1.
?a1 ssn:hasValue ?a2.
?a2 pd:readingValue ?value.

FILTER (?value > ?max)

med:Hypertension med:requiredSymptoms ?cs.
?cs med:clinicalFeatures ?cscf.

?cscf pd:hasParameter ?p.
?cscf med:clinicalRangeMax ?csrMax.
?cscf med:clinicalRangeMin ?csrMin.

FILTER ((?value > ?csrMin)&&( ?value < ?csrMax))

}
```

Looking at the two queries, it is apparent that while they do have a more complex structure, they are extracting and joining many of the same triple groups as the min/max queries. Analysing the complete set of SPARQL queries (available in appendix A) that make up the current semantic framework, it can be seen that all of the queries share the ?obs and ?range distinct triple groups.

5.4 Hadoop-Based Framework Design

With an analysis of the structure and distribution of the datasets and the required SPARQL queries completed, the next stage was to design a new algorithm that would exploit this knowledge to enable greater storage potential and performance than the current solution.

5.4.1 Required Software Functionality

The developed framework was designed with the following goals in mind:

1. The framework must perform the same functionality as the existing SPARQL queries.
2. The framework must outperform the existing Jena-based framework.
3. The framework must be scalable to cope with the potentially massive quantities of NHS data.

5.4.2 Why Use Hadoop As The Basis For The New Framework?

Section 3.2 shows that current attempts to create distributed native triplestores are flawed. They generally focus on creating systems which utilise the extra machines to allow for greater storage potential, but not extra query performance. Drawing from the literature reviewed in Chapters 2 and 3 it is apparent that Hadoop has several key theoretical advantages that make it a preferred choice over traditional and distributed triplestores for processing NHS scale data:

1. Hadoop is inherently parallel and reduces the complexity for developing a distributed application.
2. Hadoop has been shown to scale performance in a near linear fashion as the number of nodes increases.
3. Hadoop uses the HDFS which allows for the storage of data to be distributed across nodes, meaning that a Hadoop cluster can store a massive amount of data provided that the required number of nodes are present.
4. Each block of HDFS data is replicated three times across unique nodes, thus providing fault-redundant storage of medical data with no additional effort for the end user.
5. A Hadoop cluster is multifunctional and would not be limited to just one task as would be the case with a native distributed triplestore cluster.

5.4.3 The Two Approaches Taken

To assess the quality of medical data using Hadoop, two alternative Map/Reduce approaches have been created. A key consideration was the joining strategies to be used. As shown in section 3.6, the majority of the existing approaches used to process RDF data using Hadoop rely on a series of cascading reduce-side joins when processing complex queries. Due to the vast number of joins required to assess the quality of the medical data, an approach similar to the existing solutions would result in many Map/Reduce iterations needed to complete the queries. This in turn would result in very poor performance. Section 3.8 highlights

currently unexplored potential optimisations which could be exploited to create a highly efficient query system. Utilising these optimisations, for example the use of map-side and broadcast joins as well as grouping common query elements, two alternative approaches have been created. Work conducted by Myung et al. (2010) highlights N-Triple to be the most appropriate RDF serialisation format to be used for processing via Hadoop, so both the approaches will use it as the input format.

- *Approach One* - The first approach features a data upload phase, which compresses the data and then formats it in such a way as to enable efficient processing in the query stage. The query stage then performs all the required queries exploiting the structure created by the upload stage.
- *Approach Two* - The second approach features no upload stage, so the query stage is processing the raw RDF data. Due to this, its query stage is theoretically less efficient than the first approach, but does not require the potentially costly upload stage.

The following sections will explore the theoretical designs of these two approaches.

5.5 Query Planning

Before designing the implementation, a plan to complete all the queries was created. Firstly as the queries share many common elements, a super-query was created to avoid the re-computation of joins. Using this super-query removes a large percentage of duplicated joins which the standard queries would perform. The listing below shows how all the required SPARQL queries (a full list of which can be seen in Appendix A) could be compiled into a single list. While this query would not return the correct result if run upon a standard SPARQL endpoint, using Hadoop enables the correct joins to be performed. Both the Hadoop approaches collect all the elements shown below, then the required joins can be performed. These joins can re-use already completed joins and this is one of the key performance drivers of the Hadoop-based approaches.

```
?range a med:AcceptableRange .
```

```

?range med:clinicalRangeMax ?max .
?range med:clinicalRangeMin ?min .
?range pd:hasParameter ?p .

?obs a mo:PhysiologicalObservation .
?obs ssn:observedProperty ?p .
?obs ssn:observationResultTime ?time .
?obs pd:atHumanTime ?htime .
?obs ssn:observationResult ?a1
?obs ssn:observedBy ?sensor.

?a1 ssn:hasValue ?a2 .
?a2 pd:readingValue ?value .

?sensor ssn:hasMeasurementCapability ?mc.

?mc a ssn:Accuracy .
?mc ms:capabilityValue ?accuracy .

med:Hypertension med:requiredSymptoms ?cs .
med:Hypotension med:requiredSymptoms ?cs .

?cs med:clinicalFeatures ?cscf .

?cscf pd:hasParameter ?p .
?cscf med:clinicalRangeMax ?csrMax .
?cscf med:clinicalRangeMin ?csrMin .

```

Figure 5.1 shows a graphic representation of how the various triple groups are linked and how the Hadoop-based approaches process the required joins. All the elements in a triple group are linked via a single common element, with links between groups also being between common elements. Both the Hadoop approaches join the various elements contained within each triple group first and then perform the inter-triple group joins using various Hadoop joining strategies.

5.6 Approach One - Data Upload Algorithm

Before the data can be queried via Map/Reduce it first must be uploaded to the HDFS. Rather than store the data in a raw format, with no prior processing, an algorithm was designed to take the original medical RDF data, compress it and then store it on the HDFS. As upload time was one of the limiting factors in the original Jena based framework, reducing

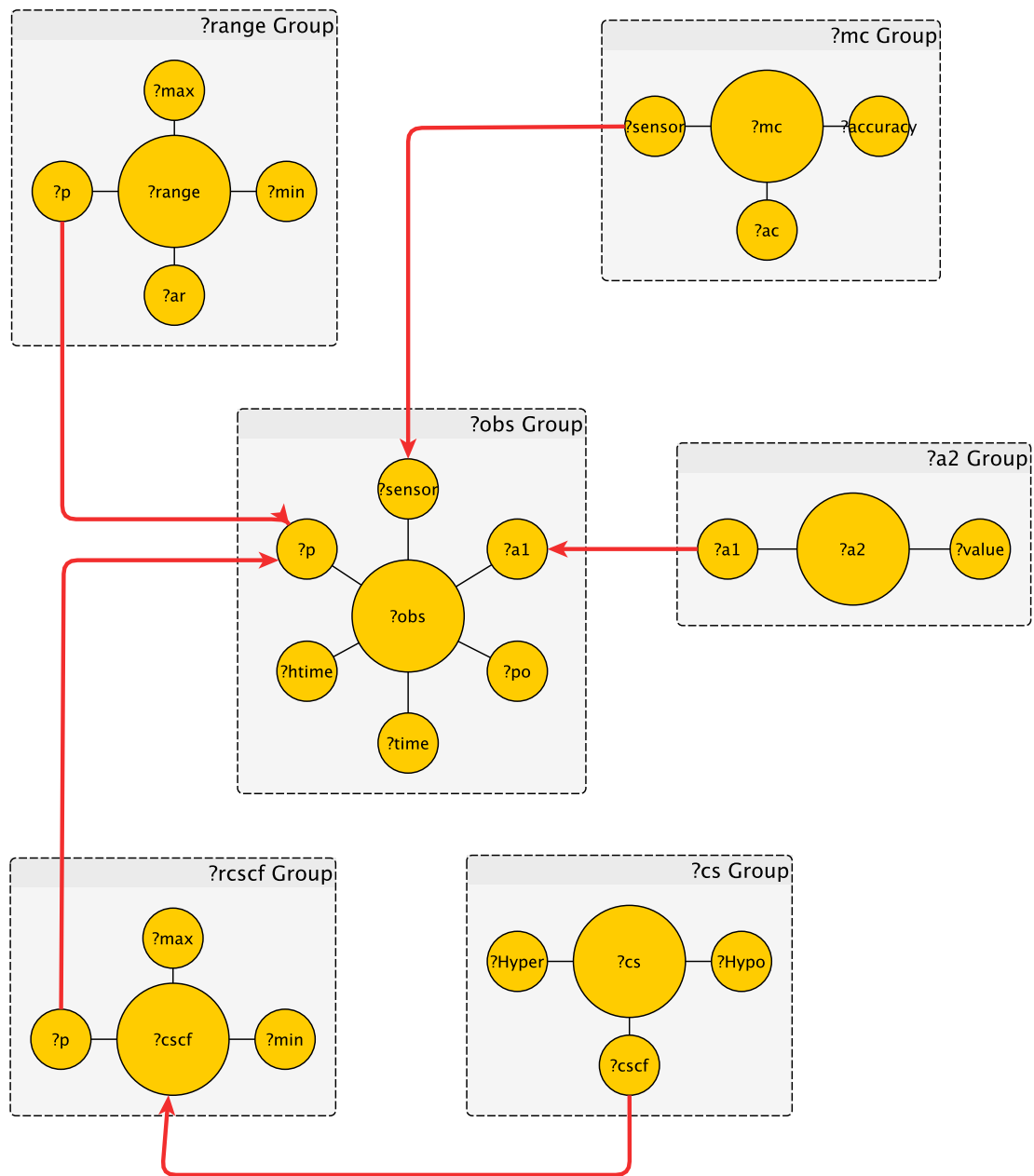


FIGURE 5.1: Triple Group Joining Plan

it was a key goal for the Hadoop implementation. The upload algorithm has been designed to both reduce the amount of space that is required to store the data on the HDFS and also format it to enable faster query processing later on.

In order to achieve this, the original RDF data is passed through two stages to first compress it and then sort it, so that all predicates and objects for a particular subject are rendered on the same line.

5.6.1 Stage One - Compression

Prior research has been performed into creating RDF compression systems using Map/Reduce, for example, dictionary encoding of RDF data using Hadoop has previously been explored by Urbani, Maassen, Drost, Seinstra, and Bal (2013). They demonstrate a system which, by using their dictionary encoding method, compresses a 92GB LUMB-based RDF dataset by a factor of three. However the system requires a slow compression and then decompression stage, both stages taking 66 minutes for the LUMB dataset.

Due to the costly nature of dictionary encoding, an alternative solution has been developed for this project. To implement this new solution, common predetermined namespaces in the original RDF data are located and replaced with shorter ones. This has the advantage of reducing the amount of space each triple consumes on the HDFS and also reducing the amount of network traffic during the shuffle and sort phase. As the namespaces to be replaced are predetermined this stage can be implemented as a Map only job. This approach was chosen instead of a full dictionary encoding as it can be accomplished in a map only job and is therefore quicker. It also requires no additional Map/Reduce stage to decode the data once complete, as is required by the dictionary encoding method.

5.6.2 Stage Two - Sort On Subject

For the second stage of the upload algorithm the compressed data is passed to a complete Map/Reduce iteration. The goal of this stage is to store all of the RDF predicates and objects for a certain subject on the same line of input on the HDFS. As shown in section 5.1, the majority of subjects used in the medical data are used in multiple unique triples. Utilising a method similar to the one described by Rohloff and Schantz (2010), can be justified both by this replication of subjects and also by the fact that many of the joins required for the SPARQL queries are performed upon the subject. Using this method reduces replication of triples to further save space on the HDFS and also enables faster performance of joins in the query stage. As many of the joins required by the SPARQL queries are acting upon common subjects, the sort on subject stage will allow the query stage to perform joins via a map-side join rather than the more costly reduce-side join method.

To implement the sort on subject stage, the Map stage scans the entire compressed RDF data set, setting each RDF subject as the key and the rest of the triple as the value. The Reduce stage then outputs the subject followed by all the associated predicates and objects. Due to the use of a reducer, the sort on subject stage has the potential to be the slower of the two stages within the complete upload algorithm.

To illustrate how this stage functions, an input of the following set of triples:

```
sub1 pred1 obj1 .  
sub2 pred2 obj2 .  
sub1 pred3 obj3 .  
sub3 pred4 obj4
```

Having been passed through the sort on subject stage, would be rendered on a single line in the format shown below. As the subject *sub1* is featured in multiple triples, the predicates and objects associated with it would be rendered on the same line.

```
sub1 pred1 obj1 pred3 obj3
```

5.6.3 Algorithm Pseudocode

The pseudocode for the complete data-upload algorithm is shown in algorithm 1. The complete source code for approach one, including the upload and query algorithm, can be found in Appendix C.

5.7 Approach One - Data Query Algorithm Design and Joining Strategies

The query algorithm is designed to replicate the results that would be returned by running the original SPARQL queries. To achieve this, several different Hadoop joining strategies were utilised along with exploitation of the distribution of the data. The query algorithm

Algorithm 1: Medical Data Upload Algorithm

```

Data: Original Medical RDF Data
1 JVM spawning initialisation stage;
2 Map Compressor (Key, Value)
3   Triple[]  $\leftarrow$  tripleParser(Value);
4   HashTable replaceSection(check, replace);
5   for number of elements in Triple do
6     | temp[]  $\leftarrow$  split Triple[i] on #;
7     | if replaceSection contains temp[0] then
8     |   | Triple[i] = replaceSection(temp[0]) + temp[1];
9     | end
10  end
11  output(Triple[0] + Triple[1] + Triple[2]);
12 end
Result: Compressed Data Stored On The HDFS
13 Map Subject Single Line (Key, Value)
14   Triple[]  $\leftarrow$  tripleParser(Value);
15   Key  $\leftarrow$  Triple[0];
16   Value  $\leftarrow$  Triple[1] + Triple[2];
17   output(Key, Value);
18 end
19 Reduce Subject Single Line(Key, Values[])
20   temp  $\leftarrow$  null;
21   for elements in Values do
22     | temp  $\leftarrow$  temp + current Values element;
23   end
24   Value  $\leftarrow$  temp;
25   output(Key, Value);
26 end
Result: All Predicates and Objects For A Set Subject Stored On Same Line

```

utilises many of the available Hadoop data joining strategies to complete the given queries in a highly optimised manner.

As discussed in section 5.1, the distribution of the medical RDF data can be categorised into two sections. Firstly a portion of the RDF triples contain information about medical equipment, sensor accuracy, permitted data ranges and medical conditions. This information is a very small proportion of the datasets and is consistent across them all. Secondly the bulk of the RDF data concerns the actual time domain records and values associated with the patients themselves. Knowledge of this meant that it was possible to join the smaller selection of triples to larger ones via a broadcast join. This was achieved by using the Hadoop feature called Distributed Cache. The Distributed Cache allows set files from the HDFS to

be pushed to either a Map or Reduce task. This allows multiple elements to be joined via a broadcast join, thus bypassing the need for a series of Cascade Reduce-Side Joins. Approach one also leverages the pre-processing of the data to allow for the use of the highly efficient map-side join to be used for a large proportion of the joins.

5.7.1 Selection Stage

This stage has two main functions and is implemented as a complete Map/Reduce iteration. Firstly, it traverses all triples stored on the HDFS and selects only those required by the queries. Secondly, it then performs some of the required joins before the data is passed to the second stage of the query algorithm. The complete workflow for the selection stage can be seen in Figure 5.2. This Figure will be referred to throughout this section.

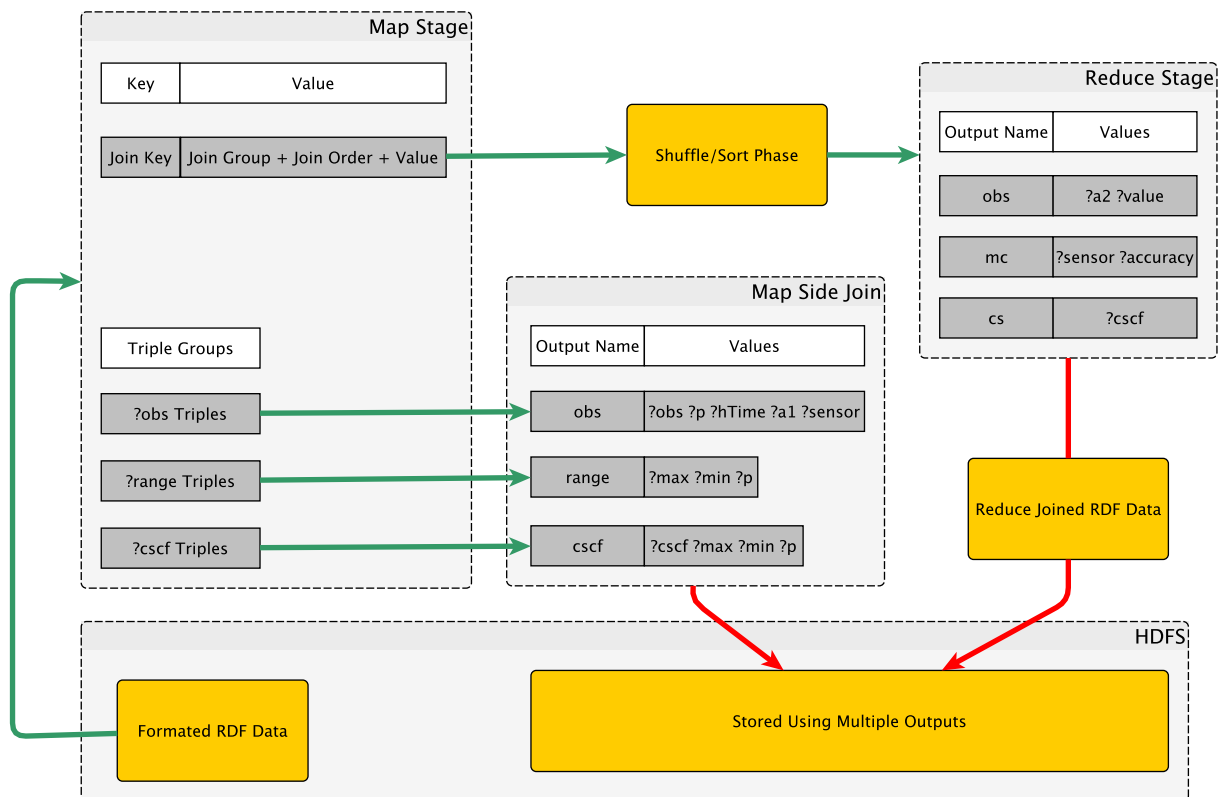
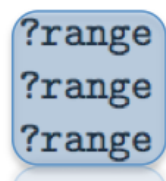


FIGURE 5.2: Approach One - Selection Stage Workflow

The input to this stage is the formatted data emitted by the upload stage and stored on the HDFS. As the data query algorithm formats the data so that all predicates and objects for

a certain subject are available on the same line, this enables a map-side join to be used to perform some of the required joins. The requirement for a map-side join is that triples needing to be joined share a common subject. As highlighted in section 3.3.1, map-side joins are the most efficient of the available Hadoop joining strategies as they bypass the need for any of the data to be transferred over the network. The map-side join is possible using this method, as all the required elements to perform the join are available from within the current map task and it requires no joins with data that is outside the reach of a certain mapper. This reach constitutes the data contained within the specific HDFS block against which the map task has been initiated.

An example section from a SPARQL query which could be joined via a map-side join is shown in Figure 5.3.



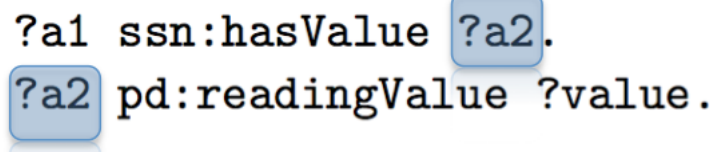
```
?range a med:AcceptableRange.  
?range med:clinicalRangeMax ?max.  
?range pd:hasParameter ?p.
```

FIGURE 5.3: Example Section Of A SPARQL Query That Can Be Joined Via A Map-Side Join

Figure 5.3 shows the `?range` portion of the query which is common across all the queries. All of the required BGPs in this portion of the query are joined on the subject. As all the predicates and objects for a certain subject are available on the same line of input, the data can be joined in the Map stage. A further optimisation implemented here is the collection of all the required values for the `?range` portion of the query in the same job. The current Jena implementation would recompute the `?range` joins each time for the minimum and maximum queries, even though the difference is one BGP. In the Hadoop implementation, both the minimum and the maximum values are collected in the same job. After the values have been joined together they are collected and then written onto separate files in the HDFS via a special output function called Hadoop Multiple Outputs. By default, any output from a Map/Reduce task is combined into a single large file. However, using Hadoop Multiple Output enables the output from any task to be split into separate, but predetermined files, when being stored on the HDFS (White, 2010).

The procedure described above is used for other triples groups which meet the requirements to be joined in the map phase. Each separate triple group is written out to its own unique file. This is done so that each triple group can be accessed directly at a later time, avoiding a costly re-search for them. The location of this process, along with the other triples groups which qualify from a map-side join can be seen in Figure 5.2. The Figure shows how the various triple groups joined via a map-side join are collected and written into separate files on the HDFS using Hadoop Multiple Outputs. This step is completed without the need to initiate a reduce task.

For joins that cannot be performed via a map-side join, they must be completed via a reduce-side join. An example case for using a reduce-side join would be when an object from one triple must be joined to a subject from a second triple. An example of this is shown in Figure 5.4.



```
?a1 ssn:hasValue ?a2.  
?a2 pd:readingValue ?value.
```

FIGURE 5.4: Example Section Of A SPARQL Query That Can Be Joined Via A Reduce-Side Join

To join the triples shown in Figure 5.4 via a reduce-side join, two stages must be performed. Firstly in the map phase, once one of the triples to be joined has been located, the element on which it is to be joined is set as the key and the rest of the triple as the value. In addition, the value is tagged with its join group and its join order. Any common elements will be passed to the same reducer via the methods discussed in section 3.3.2. The join group is used in the reduce stage to decide which elements are members of which triple groups. The join order is used to determine a set order for the elements when being written out on the HDFS. Figure 5.2 shows the location of the reduce-side join. It also shows how the final output from any reduce-side joins are also written back onto the HDFS using the Hadoop Multiple Outputs feature. This is done so that only the files which contain the relevant triples can be used as input to the join stage, thus avoiding the need to rescan the entire contents of the HDFS.

The pseudocode for the selection stage for approach one can be seen in algorithm 2. The source code for the query stage of approach one can be found in appendix C.

Algorithm 2: Medical Data Query Map/Reduce Iteration 1

```

Data: Compressed Medical RDF Data
1 JVM spawning initialisation stage;
2 Map Pass1 (Key, Value)
3   Hadoop Multiple Output Initialise;
4   dataElementList[]  $\leftarrow$  tripleParser(Value);
5   if dataElementList Can Be Joined Via A Map-Side Join then
6     if dataElementList contains required triple elements i then
7       Values  $\leftarrow$  get i1...in+1 from dataElementList;
8       output(Hadoop Multiple Output Group, Values);
9     end
10  else if dataElementList Must Be Joined Via A Reduce-Side Join then
11    if dataElementList contains required triple elements i then
12      joinKey  $\leftarrow$  get i1...in+1 from dataElementList;
13      Value  $\leftarrow$  get element to be joined plus joinGroup plus joinOrder;
14      output(joinKey, Value);
15    end
16  end
17 Reduce Reduce1 (Key, Values[])
18   Hadoop Multiple Output Initialise;
19   Create Required DataObjects;
20   for element in Values do
21     switch joinGroup do
22       case i
23         Add element to DataObject i;
24       end
25     endsw
26   end
27   if DataObject is not empty then
28     Values  $\leftarrow$  get sorted elements from DataObject via joinOrder ;
29     output(Hadoop Multiple Output Group, Values);
30   end
31 end

```

5.7.2 Join Stage

This stage has two main functions and is implemented as a complete Map/Reduce iteration. Firstly it performs the rest of the required joins to complete the queries. Secondly, it then formats and then emits the final output. The input to this stage is the data emitted by the previous selection stage. This stage makes use of the broadcast-join method discussed in

section 3.3.4. The use of a broadcast-join enables numerous elements to be joined together from within the same job. Using the existing methodology from the literature, the joins would otherwise have been performed via a series of costly cascade reduce-side joins. Figure 5.5 shows the workflow for the complete join stage Map/Reduce iteration. This Figure will be referred to throughout this section.

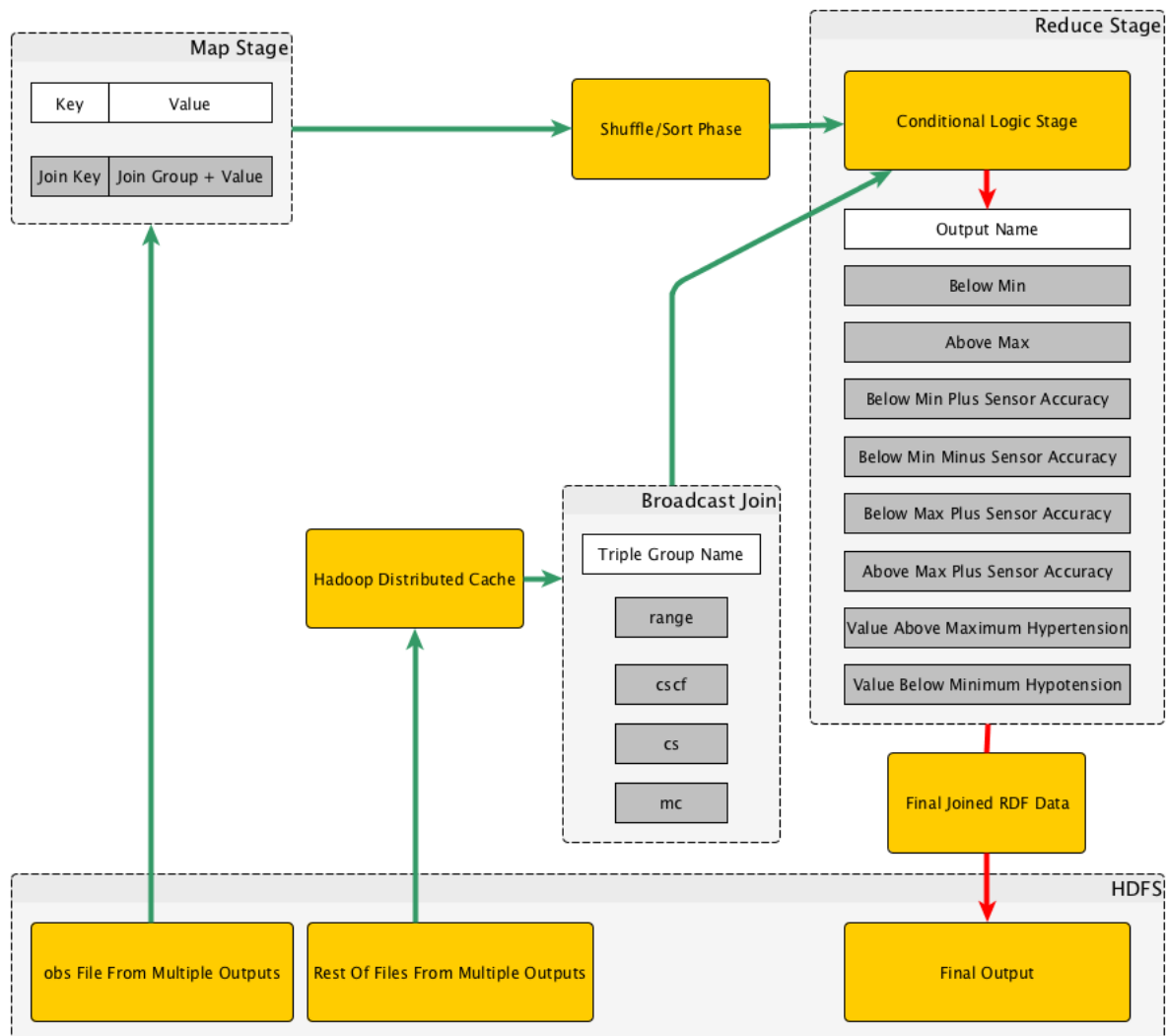


FIGURE 5.5: Join Stage Workflow

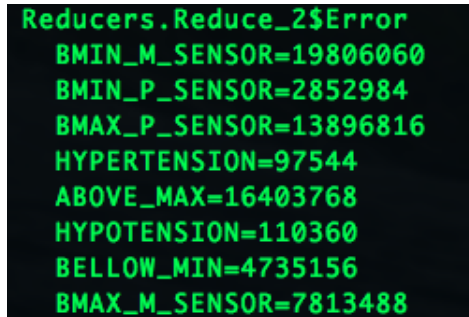
To perform the broadcast-join, the smaller files created in the selection phase are distributed to all the reducers running in the join phase. This uniform distribution of files is achieved by making use of the Hadoop Distributed Cache feature. This feature allows smaller files to be made available to any Map or Reduce task running upon any node within the cluster. The files pushed to the reduce phase are extracted into Hashtables, which allows for extremely

fast lookups when checking for element membership. The broadcast join method is particularly applicable in this case, since as highlighted in section 5.1, the data is massively skewed towards one triple group. This group, which always forms the ?obs part in any query, is far too large to be stored in memory so must be joined via a reduce-side join. However, the smaller triple groups can be joined to the larger ?obs group via numerous broadcast joins. The location of the broadcast join in the join stage workflow can be seen in Figure 5.5.

These concepts are practically implemented in the following manner. Firstly, the map phase, which takes as input a file containing the ?obs results from the reduce section. The map phase decides if the current input is an ?obs record or a ?a2 record based on length. These two triple groups are then joined via the common element and passed to the reduce phase. In the reduce phase, the two groups will be available in the same job due to the shuffle and sort phase. The system then performs the rest of the joins via the broadcast method. As explained above the files which are to be joined via the broadcast join method are all extracted from the Hadoop distributed cache and loaded into Hashtables in main memory. Then the different triple groups highlighted in Figure 5.1 can be joined via any common elements. Each one of the joins performed via the broadcast method would otherwise have to have been completed via a separate reduce-side join, as they are all performed upon individual elements. Once the required joins are completed, the algorithm then performs the conditional logic determined by original queries. The logic conditions include checking values against a pre-determined range. Figure 5.5 shows the full range of conditional logic checks which the stage performs. Once the algorithm finds a value that does not meet the requirements, it will emit the required value and the other associated values back onto the HDFS. This stage again makes use of the Hadoop Multiple Outputs, to split the output from each conditional logic check into its own file. This allows a user to more easily see and access the values which have failed a particular logic check.

In this stage, the algorithm also uses the Hadoop Counter feature. Hadoop counters are custom values that can be incremented from within either a Map or Reduce task. The counter is then recorded and output with the rest of the job completion metrics. When the algorithm has discovered data which do not meet the required quality metrics, it also increments a relevant counter. Once the algorithm has finished running, the user can assess

how many values have been judged to be errors just from viewing the final job log. An example of this output can be seen in Figure 5.6. The names represent the metrics which the framework is assessing and the numbers represent the amount of times data has failed a certain metric.



```
Reducers.Reduce_2$Error
BMIN_M_SENSOR=19806060
BMIN_P_SENSOR=2852984
BMAX_P_SENSOR=13896816
HYPERTENSION=97544
ABOVE_MAX=16403768
HYPOTENSION=110360
BELLOW_MIN=4735156
BMAX_M_SENSOR=7813488
```

FIGURE 5.6: Hadoop Job Output Showing Counters

The pseudocode for the join stage of approach one can be seen in Algorithm 3.

5.8 Approach Two

The second approach is designed to require no prior formatting of the data and takes the raw unaltered medical RDF as input. In practice, approach two is very similar to the query phase of approach one introduced in section 5.7.2. Due to the lack of prior data formatting, the use of a map-side join is not possible and the more costly reduce-side join must be used. However this drawback can be nullified due to lack of the upload phase. The second approach is again implemented as two complete Map/Reduce iterations. The source-code for approach two can be found in Appendix D.

5.8.1 Selection Phase

This stage has two main functions and is implemented as a complete Map/Reduce iteration. Firstly it traverses all triples stored on the HDFS and selects only those required by the queries. Secondly, it then performs some of the required joins before the data is passed to the second stage of the query algorithm.

The selection phase is the key difference between the two approaches, as due to lack of input formatting the phase must use the reduce-side join method to complete all joins. Figure 5.7 shows the workflow for the selection stage of approach two. Compared with the workflow for approach one, shown in Figure 5.2, it can be seen that approach two performs all of the joins in the reduce stage. This means that all of the triple groups required to complete the queries must be passed through the shuffle and sort phase and thus transferred over the network. This additional network traffic means that the selection stage of approach two should demonstrate a much worse performance than approach one.

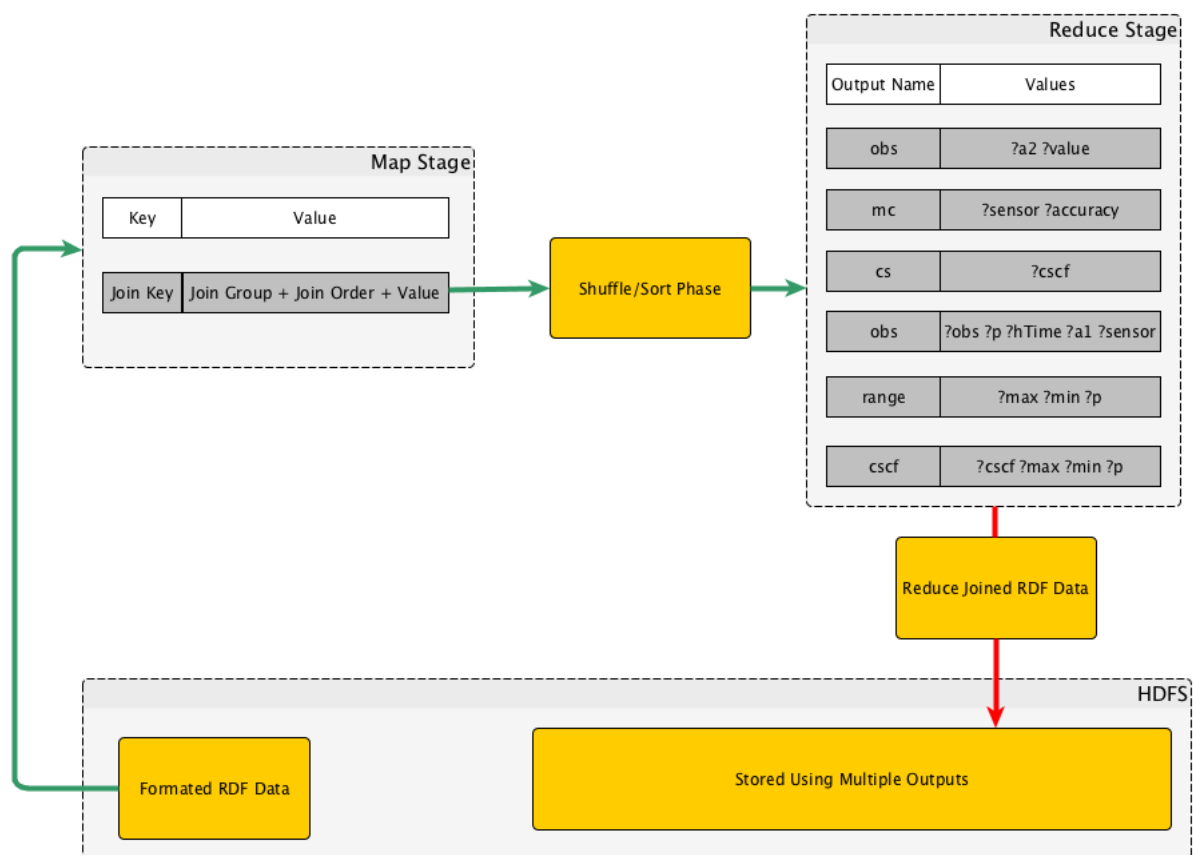


FIGURE 5.7: Approach Two - Selection Stage Workflow

5.8.2 Join Phase

The join phase for approach two is almost identical to the phase used in approach one. It joins the two largest triple groups via a reduce-side join, then joins the smaller groups via a

broadcast join using the Hadoop Distributed Cache. As the join stage for approach two is so similar to approach one, Figure 5.5 can again be used to demonstrate the workflow.

5.9 Implementation Summary

5.9.1 Key Theoretical Performance Advantages

This section will explore the key theoretical performance advantages that the two approaches offer over the Jena-based framework and also the standard method of joining data in Hadoop via a series of cascade reduce-side joins.

- Both approaches perform all the required queries and therefore joins within the same job. This saves on a massive amount of re-computation (due to common query elements) that running each query sequentially would incur.
- Approach one uses a data upload stage to format the data in such a manner as to allow highly optimal map-side joins to be utilised in the query phase.
- Both approaches use prior knowledge of the data structure to push smaller triple groups into the Distributed Cache to enable the use of the broadcast join method. This removes the need for a series of slow cascade reduce-side joins.
- The approaches should scale well as the number of nodes is increased, meaning that any additional nodes will reduce query time, not just increase maximum storage capacity as is the case with current distributed native triplestores.
- The result of any performed joins is left on the HDFS to enable any future queries to be performed more efficiently, as the joins will not need to be re-computed.

5.9.2 Summary

In summary, two different approaches to assessing the quality of medical data using Hadoop have been created. Approach one employs an upload stage to compress the data and

format it in such a way as to allow map-side joins to be used to complete a large portion of the required joins. Approach two requires no upload stage but must use reduce-side in place of map-side joins. Both approaches share the use of the efficient broadcast join which is used to avoid multiple cascade reduce-side joins. This technique has not currently been attempted by any of the current solutions presented in the literature. Both approaches perform all the required queries simultaneously. This means that the approaches avoid wasting computational resources re-performing joins on elements common across all the queries. Again this technique of grouping common query elements is not present in the current literature and is unique to this framework.

In the following sections, both the Hadoop approaches will be tested against each other and against the current Jena TDB based implementation.

Algorithm 3: Medical Data Query Map/Reduce Iteration 2

```

Data: Intermediate Files From First Iteration
1 JVM spawning initialisation stage;
2 Push Required Broadcast Join Files From HDFS Into The Distributed Cache;
3 Map Pass2 (Key, Value)
4   dataElementList[]  $\leftarrow$  tripleParser(Value);
5   if dataElementList is of size 2 then
6     joinKey  $\leftarrow$  get element0 from dataElementList;
7     Value  $\leftarrow$  get element1 from dataElementList + joinGroup;
8     output(joinKey, Value);
9   else if dataElementList is of size 5 then
10    joinKey  $\leftarrow$  get element3 from dataElementList;
11    Value  $\leftarrow$  get element0, 1, 2 and 4 from dataElementList + joinGroup;
12    output(joinKey, Value);
13 end
14 Reduce Reduce2(Key, Values[])
15   Hadoop Multiple Output Initialise;
16   Files  $\leftarrow$  Get Broadcast Join Files From The Distributed Cache;
17   Put Files Into HashMaps;
18   for element in Values do
19     switch joinGroup do
20       case i
21         Add element to HashSet ?obs or HashSet ?a2;
22       end
23     endsw
24   end
25   for element in ?obs and ?a2 do
26     Split ?obs into component elements  $\rightarrow$  (obs, p, htime, sensor);
27     Extract value From ?a2;
28     Extract min and max From rangeHashMap based on p;
29     if value outside range set by min and max then
30       output(Trigger Condition, requestedElememnts);
31       Increment Hadoop Counter For Trigger Condition;
32     end
33     Extract accuracy From mcHashMap based on sensor;
34     v2  $\leftarrow$  value * accuracy;
35     if value + v2 outside range set by min and max then
36       output(Trigger Condition, requestedElememnts);
37       Increment Hadoop Counter For Trigger Condition;
38     end
39     Extract csrMin and csrMax From cscfHashMap based on p;
40     if value outside range set by min and max combined with csrMin and csrMax then
41       output(Trigger Condition, requestedElememnts);
42       Increment Hadoop Counter For Trigger Condition;
43     end
44   end
45 end
Result: Data Which Fail To Meet Required Quality Metrics

```

Chapter 6

Test Environment and Optimisations

This chapter will detail the environments on which the Hadoop and Jena-based approaches were tested. A Hadoop cluster was created specifically for this project and so the optimisations employed in the configuration of Hadoop are also explored in this chapter.

6.1 Details Of Test Environments

The framework was tested in two separate environments: a single machine and a dedicated Hadoop cluster.

6.1.1 Single Machine

Firstly as the framework is required to be efficient even on a single machine, it has been tested on a modern, standard issue University staff desktop. The software stack for the machine comprises Fedora 20 64-Bit, Java OpenJDK 1.7.0_51, Hadoop 1.2.1 and Jena 2.11. The hardware specifications of this machine are detailed in table 6.1:

Component	Single Machine
CPU	Intel i5-3470
RAM	8GB DDR3
HDD	1TB (7200RPM)

TABLE 6.1: Specification Of The Single Machine

On this machine Hadoop was configured to run in a pseudo-distributed manner. This means that the machine is acting as both the head node and also as a data node. To run the experiments on Jena 2.11, the tdbloader and tdbquery unix binaries were used. The output from these programs were written to disk so that a fair comparison with Hadoop framework could be drawn.

6.1.2 Dedicated Hadoop Cluster

Secondly the scalability of the solution was tested against massive RDF datasets on a dedicated Hadoop cluster. The cluster comprises a head node with eight data nodes, all of which contain commodity off-the-shelf components. All of the machines are running an identical software stack, comprising CentOS 6.5 64-Bit, Java OpenJDK 1.7.0_51 and Hadoop 1.2.1. All the machines communicate via a dedicated Gigabit switch. The hardware specification of the cluster nodes is detailed in table 6.2.

Component	Head Node	Data Node
CPU	Intel Q8400	Intel Q8400
RAM	4GB DDR2	8GB DDR2
HDD	250GB (7200RPM)	250GB (7200RPM)

TABLE 6.2: Specification Of The Hadoop Cluster

6.2 Hadoop Cluster Performance Optimisations

By default, Hadoop will provide a configured install which is not optimised to the cluster on which it is running. The configuration of Hadoop is controlled via a series of XML files which need to present on all machines within the cluster. The key configuration files are shown in Figure 6.3 (White, 2010).

According to White (2010) there are several parameters contained within these files that are key for optimising Hadoop for the cluster. To maximise the performance of the framework, several optimisations were tested and then implemented to tune Hadoop to give optimum performance.

Filename	Description
core-site.xml	General configuration settings for Hadoop that are common to both the HDFS and MapReduce.
hdfs-site.xml	Configuration settings which control the three HDFS daemons: namenode, secondary namenode, and datanode.
mapred-site.xml	Configuration settings which control the two MapReduce daemons: jobtracker, and tasktrackers.

TABLE 6.3: Hadoop Configuration Files

6.2.1 Benchmark Generation

In order to test the effectiveness of any configuration changes to the cluster, benchmark results were collected. Rather than use the standard Hadoop benchmark, one was designed that would reflect the type of operations that would be performed by the framework. These include moving files on the HDFS, passing data between reduce tasks and both Map only and Map/Reduce jobs. Configuration changes were then assessed for performance improvement using this benchmark. All benchmarks were repeated five times and an average taken to compensate for time variations.

6.2.2 Optimum Number Of Map/Reduce Tasks

By default, Hadoop only spawns two Map tasks and two Reduce tasks per node, however according to White (2010) the combined number of tasks should outnumber the amount of processors on each datanode by a factor of between one and two. To assess how increasing the number of Map/Reduce tasks affects the real world performance of the cluster, a number of different values were tested. Values of two, three, four and six Map/Reduce tasks were used. The results of the benchmark can be seen in Figure 6.1.

As the results show, changing the number of Map/Reduce tasks does have an impact on Hadoop's performance. The default value of only two Map/Reduce tasks consistently performs the worst across all five runs. Increasing the number of tasks to three and then four increased performance over two tasks. When the number of tasks was increased to six, performance began to decrease again. This decrease in performance was expected as all

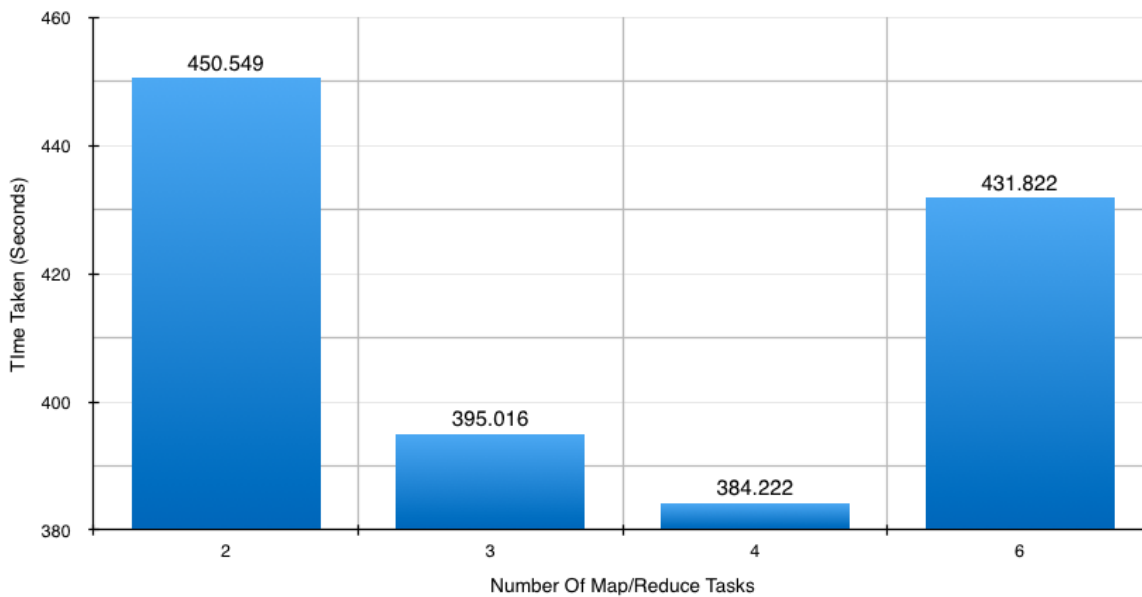


FIGURE 6.1: Optimum Number of Map/Reduce Tasks Results

the datanodes only have quad-core processors. Overall the results show that the optimum number of Map/Reduce tasks for the cluster is four. This result reflects the theory which states that the number of Map/Reduce tasks should outnumber the amount of processors on each datanode by a factor of between one and two. The final value used for the number of Map/Reduce tasks is four.

6.2.3 Optimum JVM Heap Memory Allocation

White (2010) provides a formula, shown in Figure 6.2 which can be used to calculate Hadoop's memory footprint. By default, Hadoop only allocates 200MB of heap memory to each task. According to White (2010) increasing this to efficiently utilise all available memory is a key performance enhancement.

To assess how increasing the heap size assigned to each task affects the real-world performance of the cluster, a number of different values were tested. Values of 200, 400, 600 and 800 MB were used. The results of the benchmark can be seen in Figure 6.3.

The results show that the default heap size of 200MB used by Hadoop is consistently the worst performing configuration. Increasing the heap size to 400MB improves performance

JVM	Default memory used (MB)	Memory used for 8 processors, 400 MB per child (MB)
Datanode	1,000	1,000
Tasktracker	1,000	1,000
Tasktracker child map task	2×200	7×400
Tasktracker child reduce task	2×200	7×400
Total	2,800	7,600

FIGURE 6.2: Hadoop Memory Footprint Calculation (White, 2010).

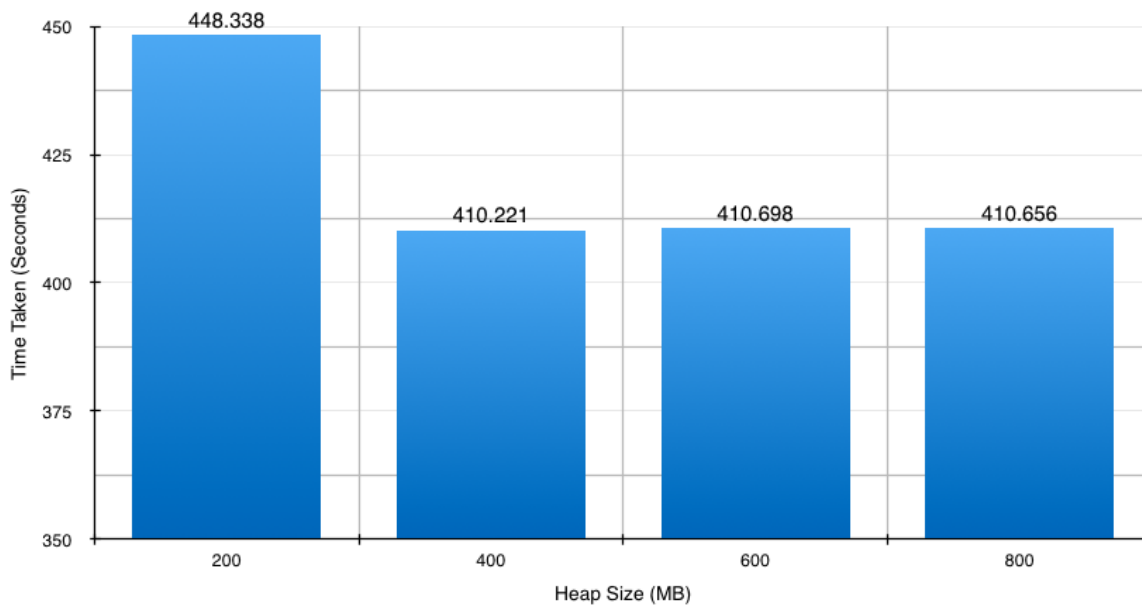


FIGURE 6.3: Optimum Heap Size Result

by a small margin over a heap size of 200MB. Increasing the heap size to 600MB and then to 800MB neither increases nor decreases cluster performance. This result could highlight that the type of operations performed by the framework do not benefit from having extra heap space assigned to tasks.

Taking the results into consideration and to efficiently utilise all of the memory, each Map and Reduce task was given 600MB of heap space. Table 6.4 shows how memory in the cluster datanodes is utilised, with the entire collection of Hadoop services consuming a total of 6,800MBs. Each datanode has 8GB of RAM, which leaves over 1GB of RAM to be reserved for the OS.

JVM	Memory Used (MB)
Datanode	1000MB
Tasktracker	1000MB
Tasktracker child map task	4 X 600MB
Tasktracker child reduce task	4 X 600MB
Total	6800MB

TABLE 6.4: Hadoop Memory Allocation on Cluster

6.2.4 JVM Re-use

As discussed in section 2.5.1, each Hadoop service runs as a separate JVM. According to White (2010), there is an overhead to start the JVMs which runs the Map and the Reduce tasks. White (2010) explains that Hadoop enables the re-use of JVMs which can lead to an increase in performance by removing the need for multiple and costly initialisation phases.

To assess how allowing the re-use of JVM affects the real-world performance of the cluster, it was benchmarked with JVM use disabled and then enabled. The results of the benchmark can be seen in Figure 6.4.

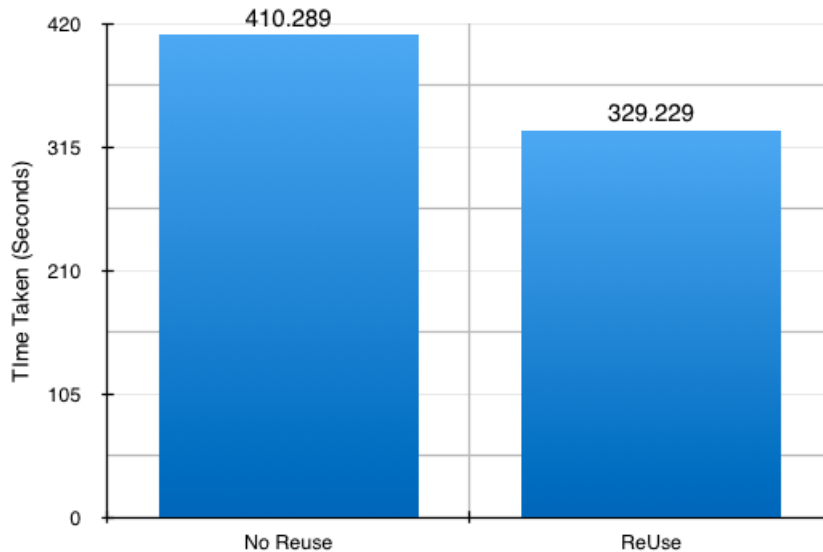


FIGURE 6.4: JVM Reuse

As the results show, allowing the reuse of JVMs has a large impact on cluster performance. In every instance, jobs in which JVM reuse was allowed performed the benchmark faster. Over the five runs, JVM re-use was an average of 40 seconds faster.

6.2.5 Sort Memory Allocation

According to White (2010), Hadoop only allocated 4KB of memory for its input/output operations. This allocation can be too conservative for a cluster comprising modern hardware and White (2010) recommends increasing this to 128KB. To test whether increasing to this value improved performance, the default value of 4KB was benchmarked against the recommended value of 128KB. The result of this can be seen in Figure 6.5.

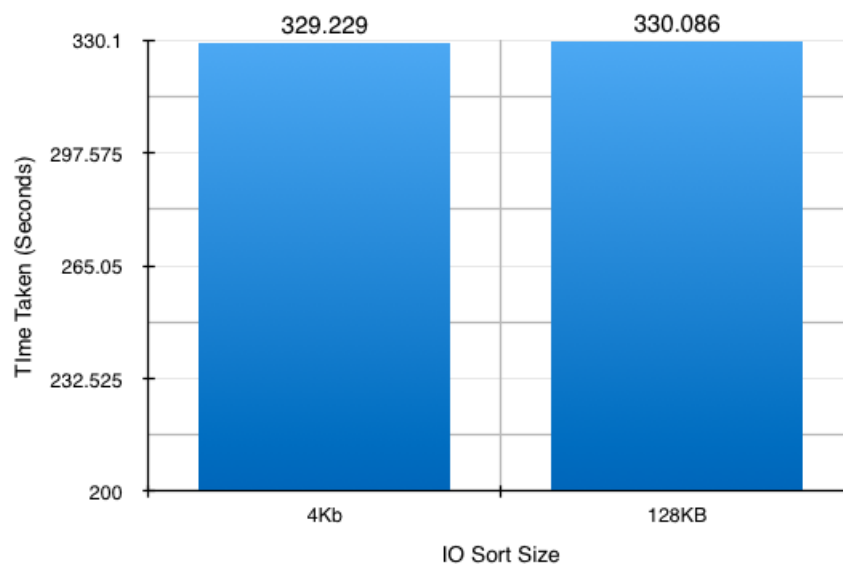


FIGURE 6.5: Hadoop Sort IO Result

The results show that increasing the sort memory on average has no real impact, either negatively or positively, on cluster performance. The increase in sort memory from 4KB to 128KB leads to an average drop in performance of just 0.8 seconds.

6.2.6 Final Configuration Values

All the final configuration values used for the cluster are shown in Table 6.5. All configuration values were driven by the results of the benchmark. All of the final Hadoop configuration XML files used for the cluster are located in Appendix B.

Configuration	Value
Num of Map/Reduce Tasks	4
Map/Reduce Heap Memory	600MB
JVM Reuse	Allowed
Sort IO Allocation	128KB

TABLE 6.5: Hadoop Cluster Final Configuration Values

Chapter 7

Results

7.1 Testing Methodology

In this chapter, the two different Hadoop-based approaches will be benchmarked when running on a dedicated Hadoop cluster and a single machine. The two approaches will also be compared directly against Jena and then against a naïve implementation to assess how successful the approaches are in increasing performance and scalability. The input datasets for all the tests comprised variable amounts of the generated RDF data, stored in N-Triple format. The results will be presented in the following manner: firstly, results from running on a single machine will be shown, followed by results from running on a dedicated Hadoop cluster.

In this context, a result is the time taken at the end of a successfully completed job, subtracted from the time at the start. These values are generated from within the Hadoop code itself and incorporate all the JVM initialisation and HDFS write stages. This gives the total overall run time of a job. For the Jena-based results, the collection of the start and end time values as well as the running of the Jena binaries was performed by a shell script.

All the experiments were repeated five times and an average taken to produce the final presented results. Each test was repeated to compensate for any variations in run time, stemming from background operating system or Hadoop health tasks. Five was chosen as the appropriate number of repeats for the experiments as it judged was the best compromise

between the limited time available for the project and running an appropriate number of repeats.

7.2 Single Node Results

In order to assess how the Hadoop-based approaches compare directly with the Jena-based implementation, both were tested against a range of RDF dataset sizes when running on a single machine. The tests were performed on the same machine described in section 6.1.1, so direct comparisons could be drawn. Results will show the total average performance for the different algorithms, along with a more detailed breakdown of how performance is spread between the underlying Map/Reduce iterations. The range of dataset sizes used for the test was: 4M, 8M, 16M, 32M, 64M, 128M, 256M, 512M and 1000M RDF triples.

7.3 Single Node Results - Approach One

7.3.1 Upload Results

Figure 7.1 shows the average time taken to perform the two stages of the upload algorithm. It shows that as dataset size doubles, the time taken to process increases by an average factor of 2.2. The increase from 512M to 1000M triples has a disproportionately large increase in process time. The jump to 1000M triples took three times longer to process than 512M did.

7.3.2 Upload Results Showing Breakdown Of Passes

Figure 7.2 displays a breakdown of how long each of the two passes (which together form the upload algorithm) took to complete. The Figure shows that pass two takes the majority of the time taken to complete the upload stage. Pass two is the stage in which all the predicates and objects for a set subject are rendered to a single line. This stage involves a

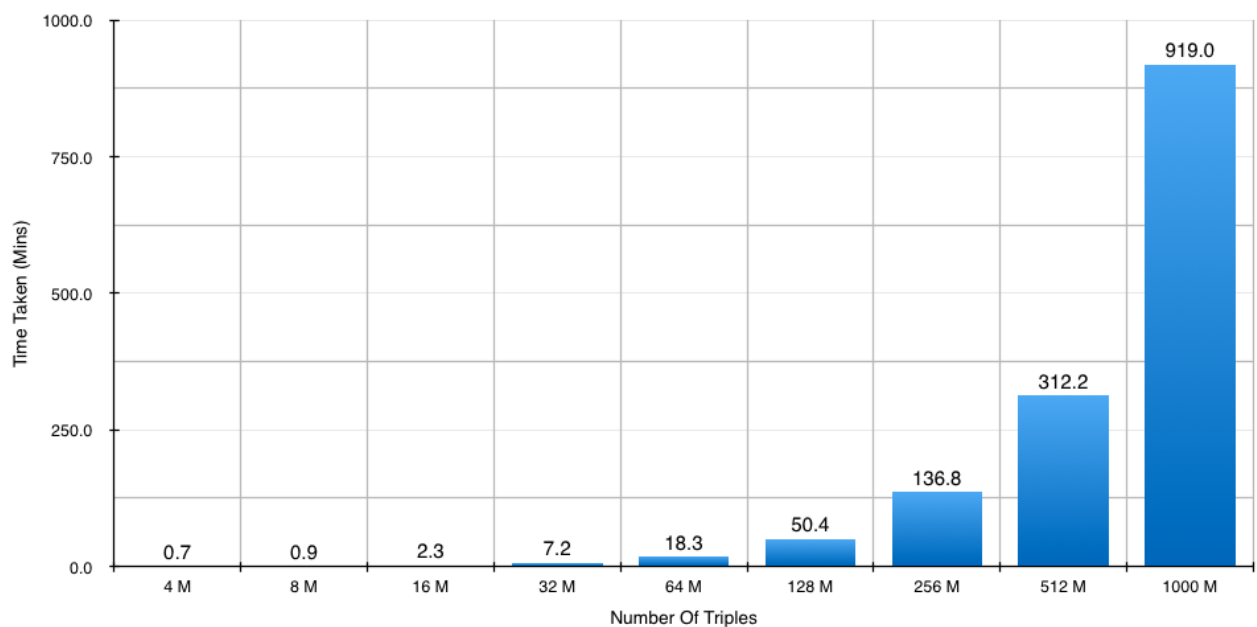


FIGURE 7.1: Approach One - Single Machine Upload Time

reduce stage so it must transfer a lot of data over the shuffle and sort phase, thus decreasing performance.

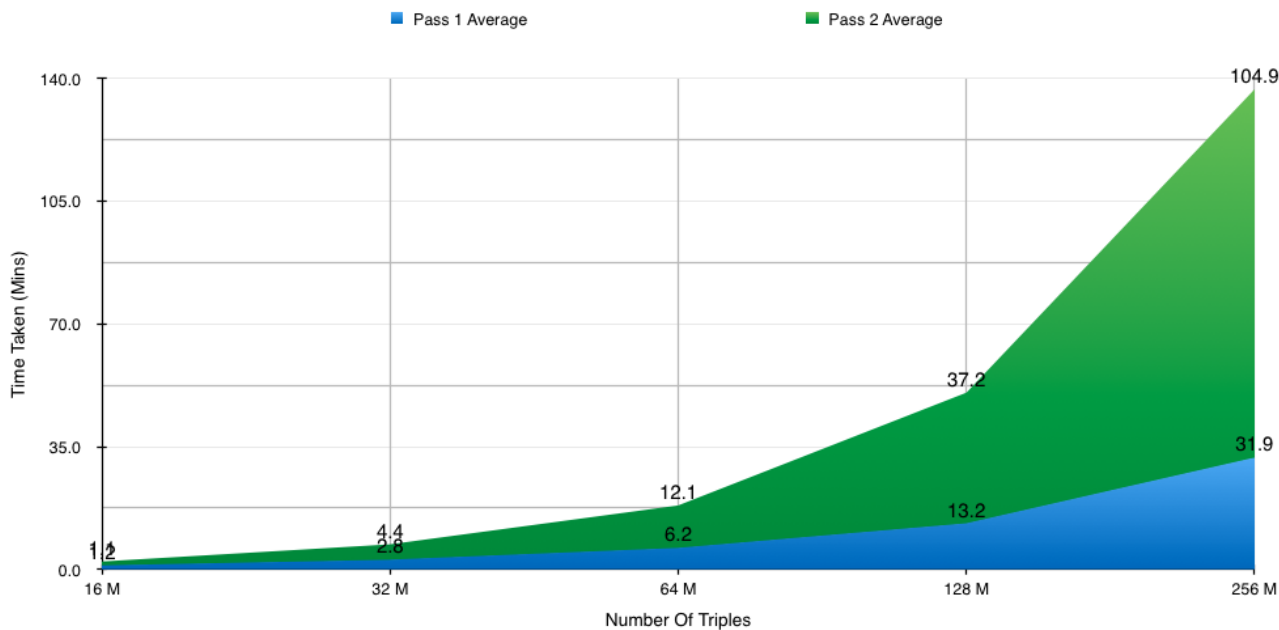


FIGURE 7.2: Approach One - Single Machine Upload Time Showing Breakdown Of Passes

7.3.3 Query Results

Figure 7.3 shows the average time taken for the framework to perform all the required queries. It shows that the framework scales extremely well as dataset size increases. Approach one is able to complete all the queries (Found in appendix A) and the associated joins for 256M triples in under 30 minutes. However at a dataset size of 1000M triples the time taken to query is abnormally large and does not follow the performance characteristics set by the preceding dataset sizes. As each query was repeated five times and an average taken, this result cannot be explained as being an anomaly. Indeed, it must represent the performance limit of the current solution on the standard desktop machine used for this test.

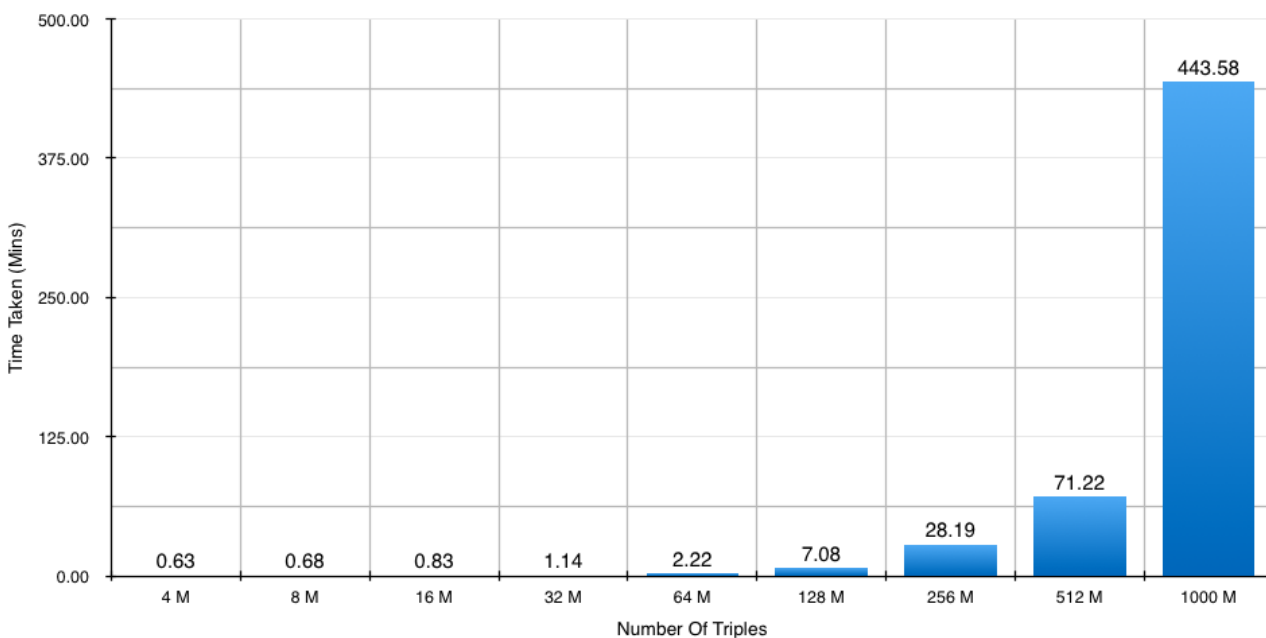


FIGURE 7.3: Approach One - Single Machine Query Time

7.3.4 Query Results Showing Breakdown Of Passes

Figure 7.4 shows a more detailed breakdown of how the query algorithm performs. For each dataset size it shows the average time taken for Pass 1 (the selection stage) and Pass 2 (the join stage) and the combined total. The Figure shows how Pass 1 is consistently the stage taking the greatest amount of time. This is the stage in which the entire collection

of triples is traversed and the relevant ones are extracted, so the assumption can be made that this stage has the largest data size input. Stage two is where the majority of the joins are performed but it is also the best performing stage.

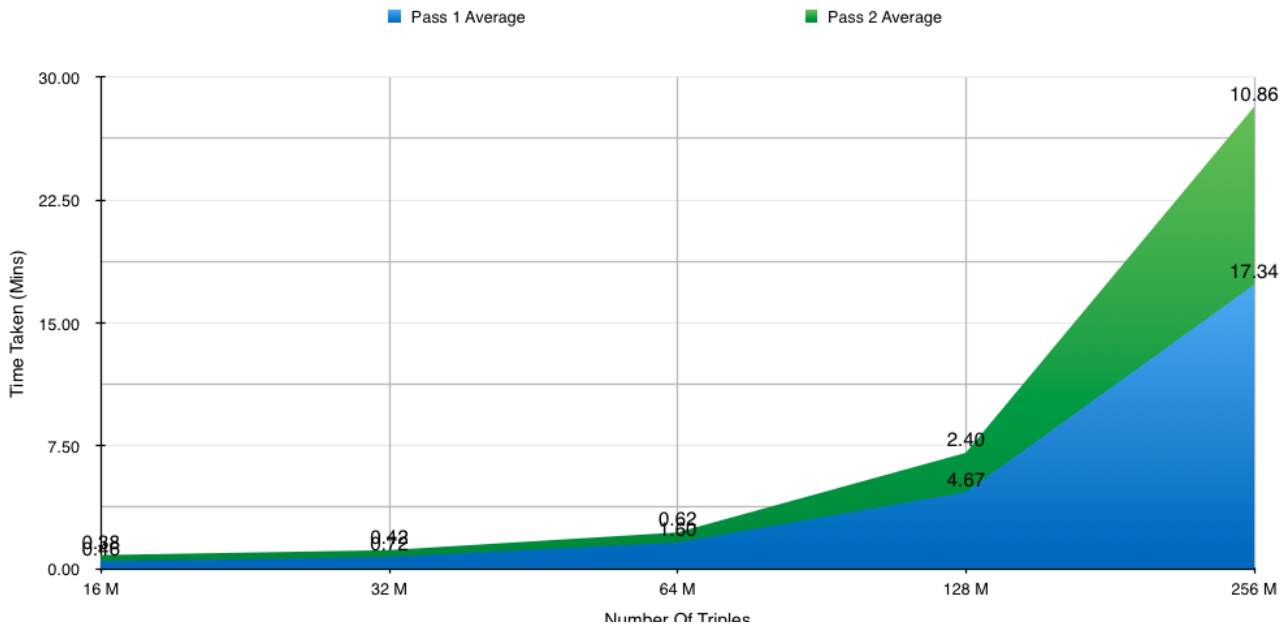


FIGURE 7.4: Approach One - Single Machine Query Time Showing Selection And Join Stages

7.4 Single Node Results - Approach Two

7.4.1 Query Results

Figure 7.5 shows the time taken to query the data using the second approach. As highlighted in section 5.8, this approach features no prior formatting of the data and uses the more costly reduce-side method to complete many of the joins. The results show that approach two scales well as the dataset size increases, although as with approach one, the time taken to process 1000M triples is high.

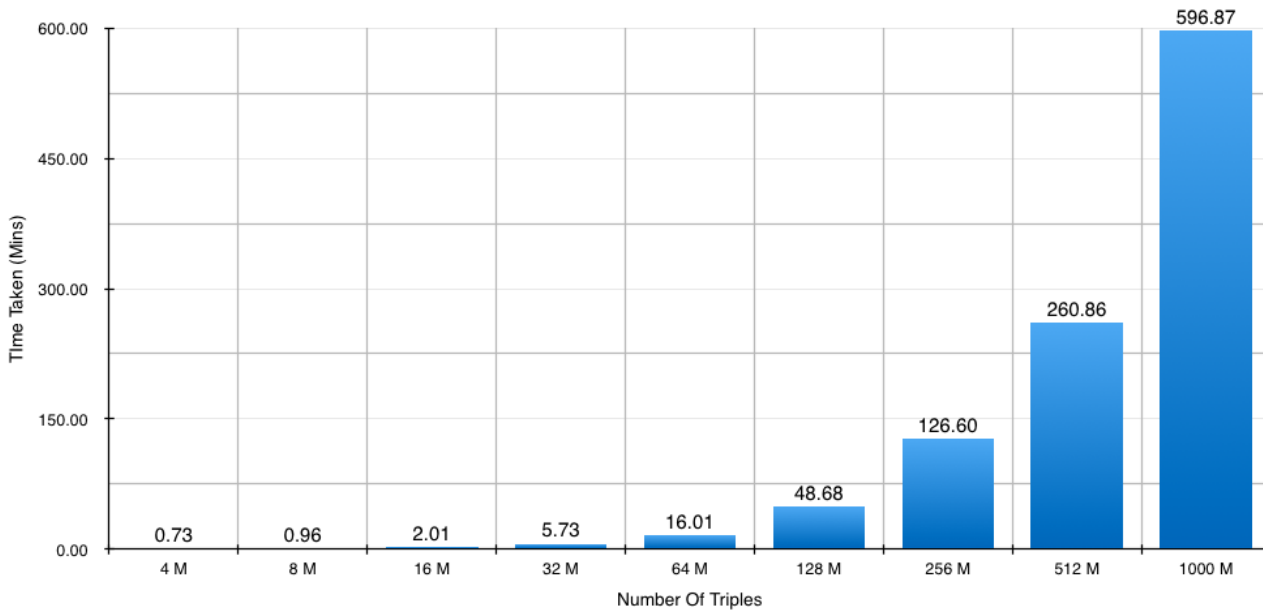


FIGURE 7.5: Approach Two - Single Machine Query Time

7.4.2 Query Results Showing Breakdown Of Passes

Figure 7.6 shows the breakdown for the two passes that together comprise approach two. Mirroring the results for approach one, the selection stage is consistently the slower of the two stages. As expected the selection stage for approach two takes a greater amount of time than that for approach one. This is because approach two must perform all of the required joins in the reduce stage, whereas approach one performs many joins in the map stage.

7.5 Single Node Comparison

7.5.1 Approach Comparison

This section presents a performance comparison between the two approaches when running on a single node. Figure 7.7 shows that approach one is the better performing of the two approaches when considering only the query stage. For the majority of dataset sizes, approach one is substantially faster than approach two. As an example for 256M triples

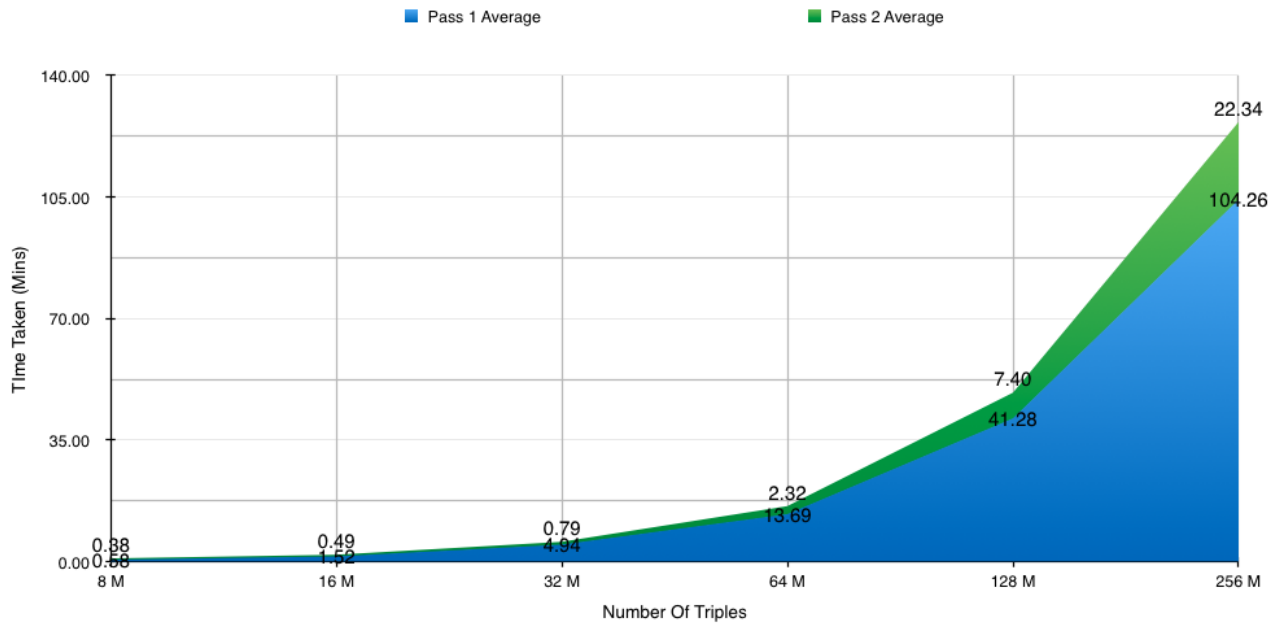


FIGURE 7.6: Approach Two - Single Machine Query Time Showing Pass Breakdown

approach one completes all the queries in 28.19 minutes, while approach two took 126.6 minutes. This means that there is a 349% increase in time taken to complete a query going from approach one to two. This difference in performance can be explained by approach one processing compressed data and joining using a map-side join. Indeed the large difference in performance demonstrates just how effective this technique is in reducing the query time.

For both approaches, a data set size of 1000M triples returns a query time which does not follow the performance characteristics of the rest of the dataset sizes. This reinforces the notion that 1000M is in practice the performance-ceiling for the standard staff desktop.

To more directly compare the two approaches, the upload algorithm from the first approach also needs to be included to allow for a fairer comparison between the total time required. A comparison between the combined time taken for the upload and query sections of approach one and approach two can be seen in Figure 7.8. The Figure shows that when the complete algorithms are compared, approach two has the best performance across all dataset sizes. This illustrates that the upload stage of approach one is very costly in terms of performance.

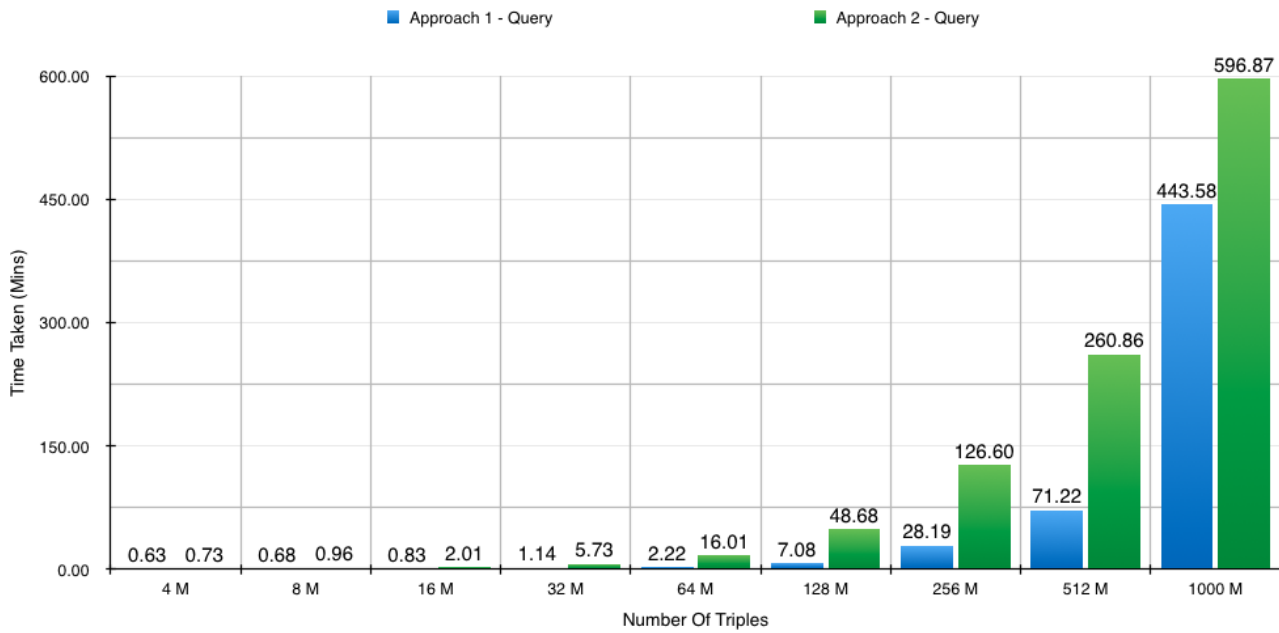


FIGURE 7.7: Single Node Query Performance Comparison Between Approaches

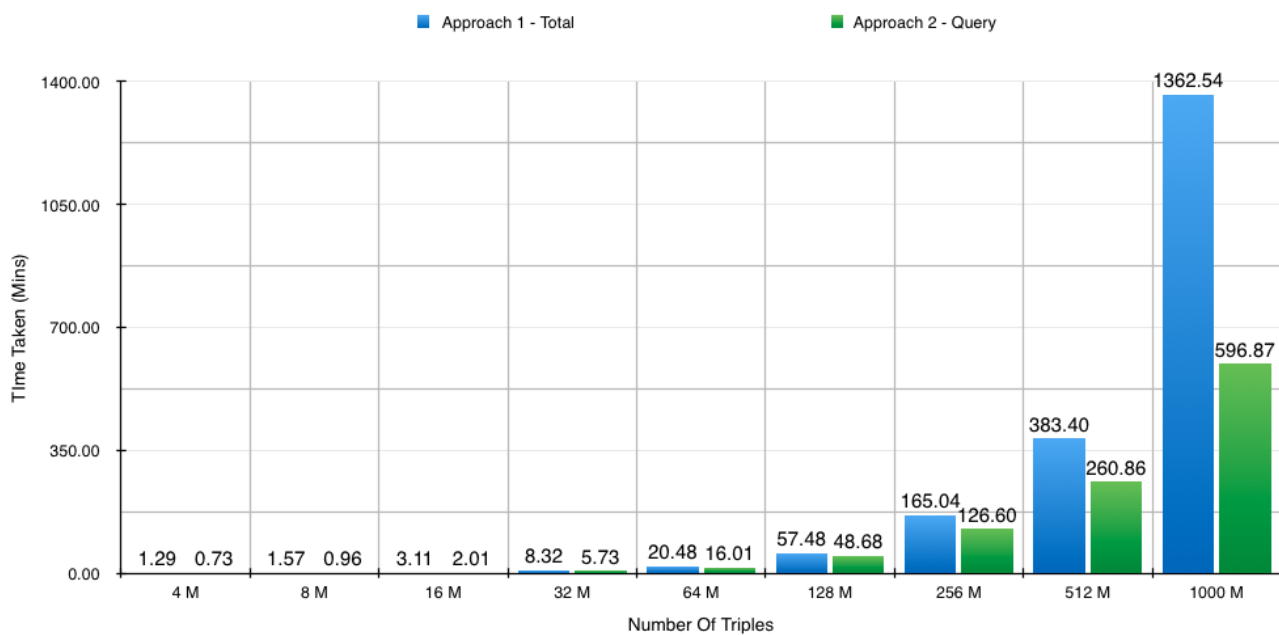


FIGURE 7.8: Single Node Total Performance Comparison Between Approaches

7.5.2 Comparison With Jena TDB

Figure 7.9 shows how approaches one and two compare directly with Jena TDB, running against the same dataset sizes and on the same machine. The results show the total

time taken for all the approaches, this means that for approach one and Jena the results in Figure 7.9 show the combined upload and query time. This shows that both the Hadoop-based approaches perform better than Jena across all dataset sizes. Another important consideration is that Jena was unable to return a result when tested again a dataset size of 128M and above, while both the Hadoop approaches were able to cope with dataset sizes up to 1000M triples. However, the Hadoop approaches not only demonstrate better performance, but they are also more scalable across dataset sizes, even when running on a single node.

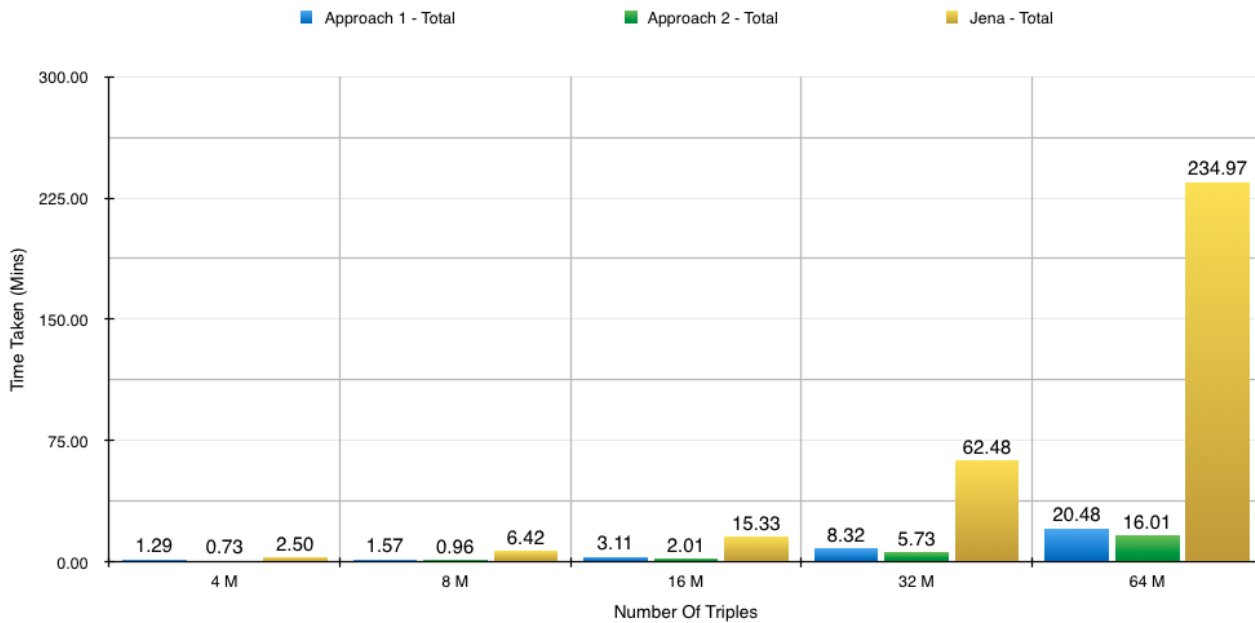


FIGURE 7.9: Single Node Total Performance Comparison Between Hadoop And Jena

This section has demonstrated that Hadoop is more suited to processing NHS-scale RDF data than Jena TDB, even when running on a single node. However the real strength of Hadoop is its ability to distribute workload across a cluster of compute nodes. The following sections explore how the performance of the two Hadoop-based approaches is affected when they are run on a dedicated Hadoop cluster.

7.6 Hadoop Cluster Results

The Hadoop-based framework was tested on the dedicated Hadoop cluster across a range of node numbers and dataset sizes. Details of the dedicated Hadoop cluster can be found in section 6.1.2. The framework was tested on one, two, four and eight nodes and across a dataset size range of 32M, 64M, 128M, 256M 512M and 1000M triples.

7.7 Cluster Results - Approach One

7.7.1 Upload Performance Across Eight Nodes

Figure 7.10 shows the upload performance of approach one across eight nodes of the cluster. Mirroring the performance characteristics from the single node, the results show that pass two constitutes by the far the larger proportion of the total time taken to upload the data.

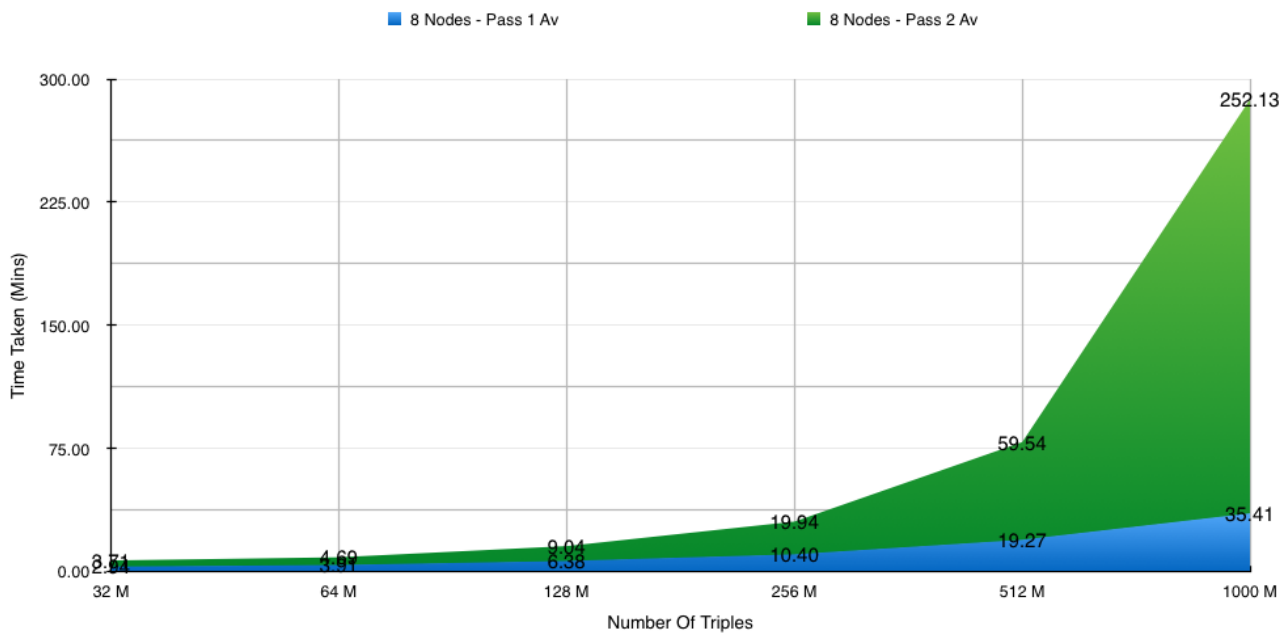


FIGURE 7.10: Approach One - Upload Performance Across 8 Nodes

7.7.2 Query Performance Across Eight Nodes

Figure 7.11 shows the query algorithm performance across eight nodes of the cluster. The results show that the cluster of eight nodes can complete all the queries across a billion triples in 87 minutes. Hence, once again the selection stage constitutes the vast majority of the total time taken to query the data. The cluster results show again how optimised the join stage of approach one is, with the eight node cluster being able to perform all the joins required for one billion triples in under eight minutes.

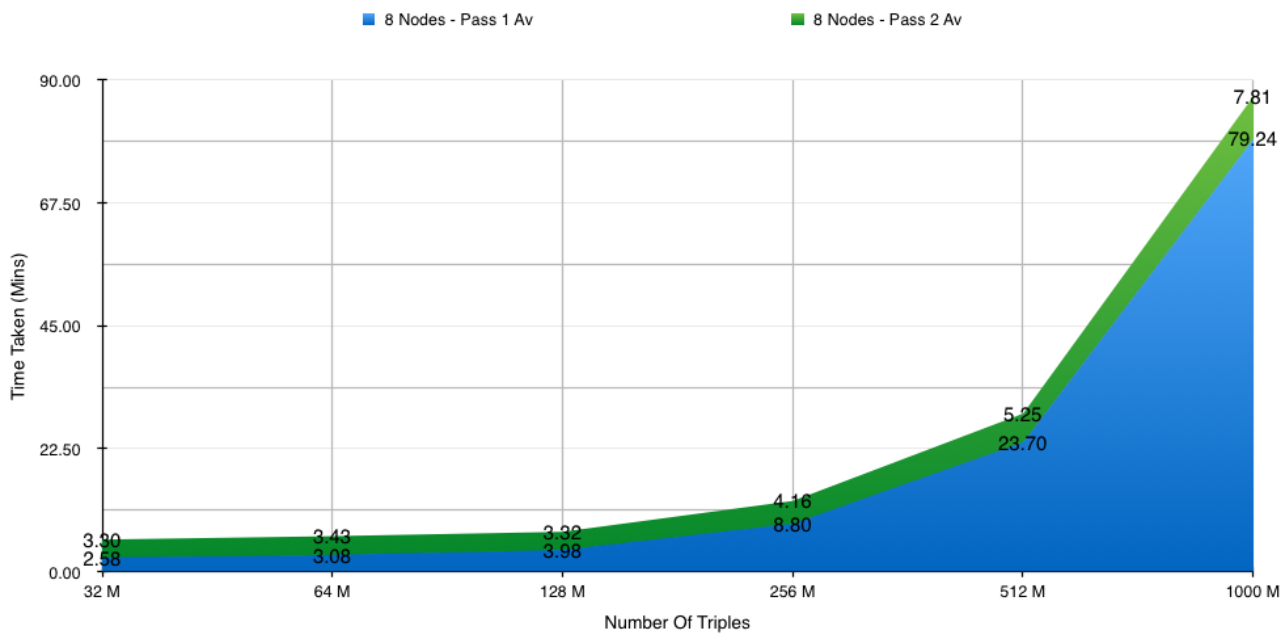


FIGURE 7.11: Approach One - Query Performance Across 8 Nodes

7.7.3 Performance Scalability Across Number Of Nodes

To test how approach one scales across different cluster sizes, it was run on different numbers of nodes. The results for 512M and 1000M are absent for the single node as the volume of storage space required to store these volumes of triples was greater than the 250GB of HDFS capacity available on the single node.

Figure 7.12 shows how the upload algorithm of approach one scales across nodes. As the number of nodes increases, the time taken to complete the upload algorithm decreases. For

example it took the single node cluster configuration 275 minutes to upload 256M triples, while it only took the eight node cluster configuration 30 minutes to process the same number of triples.

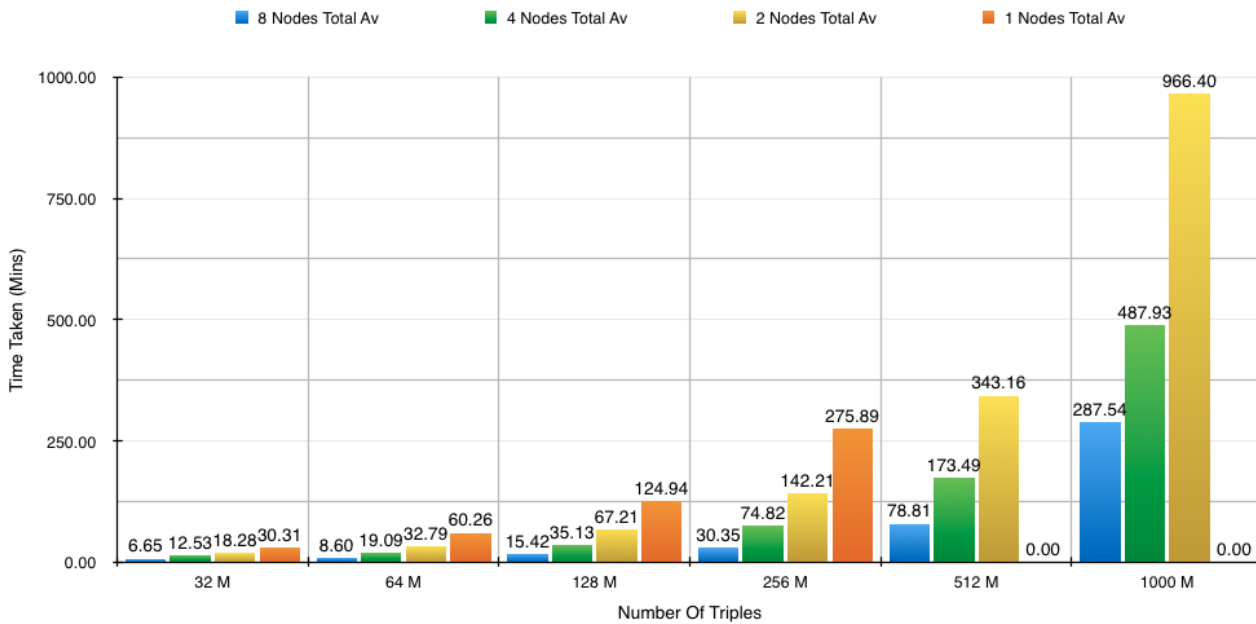


FIGURE 7.12: Approach One - Upload Performance Scalability Across Nodes

Figure 7.13 shows how the query algorithm of approach one scales across different numbers of nodes. Again, increasing the number of nodes has a dramatic effect on the total time taken to complete the query algorithm. The increase in number of nodes always results in a decrease in total time taken to complete the collection of queries.

7.7.4 Speed-Up Analysis Of Approach One's Query Stage

To assess the relationship between number of nodes and run time, a comparison of speed-up was performed. To create this comparison, the total time taken for the query algorithm of approach one across two, four and eight nodes was divided by the time taken for one node. This enables the speed-up resulting from increasing the number of nodes to become apparent. The best-case scenario would be that the speed-up would reflect the increase in node numbers. For example, this best-case scenario would mean that time taken to complete a set task on one node would be eight times faster when running on eight nodes.

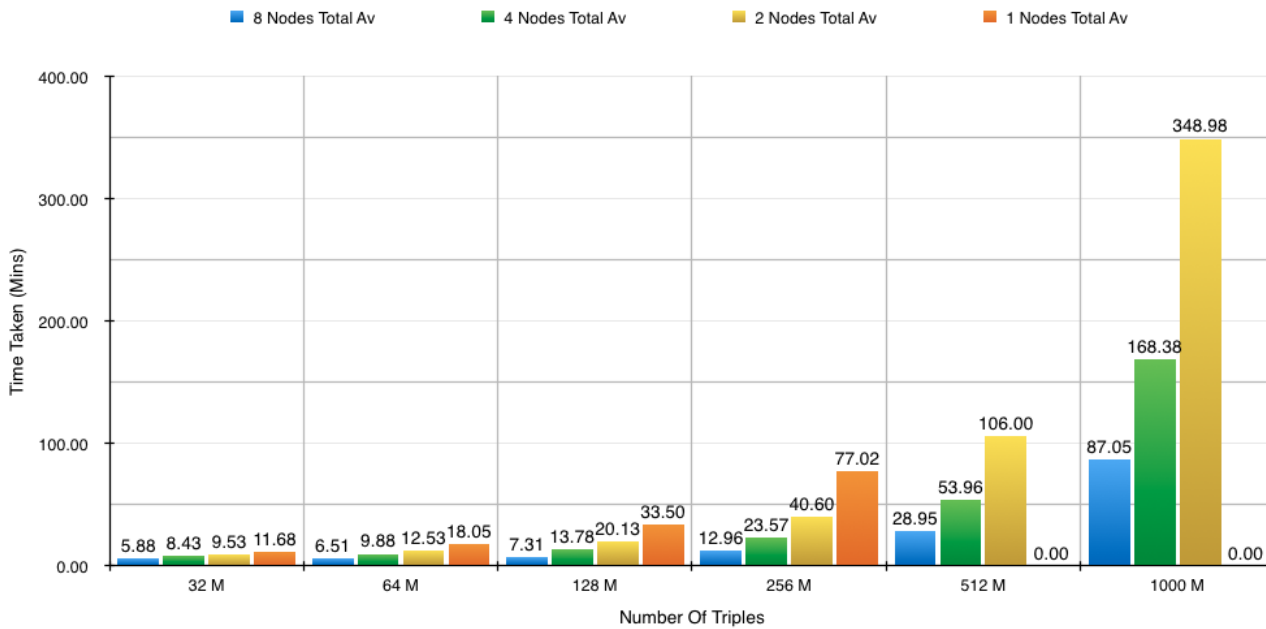


FIGURE 7.13: Approach One - Query Performance Scalability Across Nodes

Figure 7.14 shows the speed-up of different node numbers against a single node on dataset sizes up to 256M. The results highlight some very interesting trends in the query speed-up of approach one. The speed-up factor increases consistently as the number of nodes increases, which is to be expected. However the speed-up factor for a particular number of nodes does not stay consistent across dataset sizes. For example at 32M triples the speed-up factor of going from one node to eight nodes is 2, however at 256M triples the speed-up factor is 5.9.

7.8 Cluster Results - Approach Two

7.8.1 Query Performance Across Eight Nodes

Figure 7.15 shows the query performance of approach two across the 8 node cluster. As discussed in section 5.8, this result represents the total time taken for approach two as there is no upload stage. The Figure shows how pass one (the selection stage) constitutes the vast majority of the total time taken. When compared to the selection stage of approach

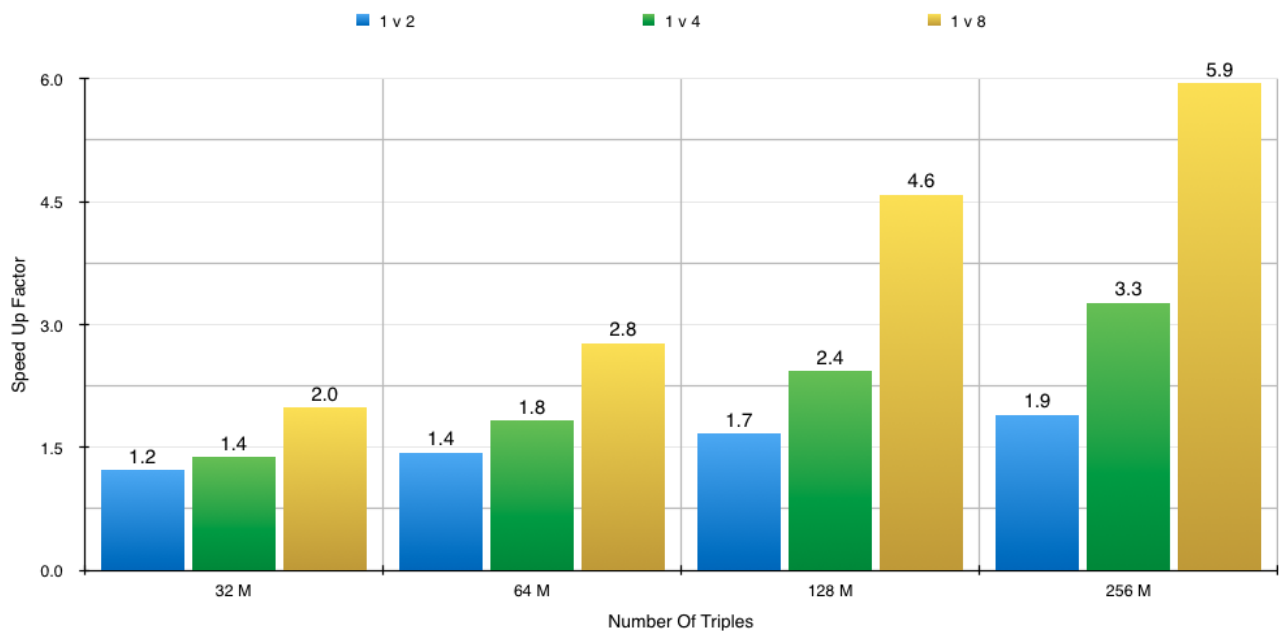


FIGURE 7.14: Approach One - Query Speed-Up Factor

one, shown in Figure 7.11, it can be seen that approach two spends comparatively more of its total run time completing the selection stage.

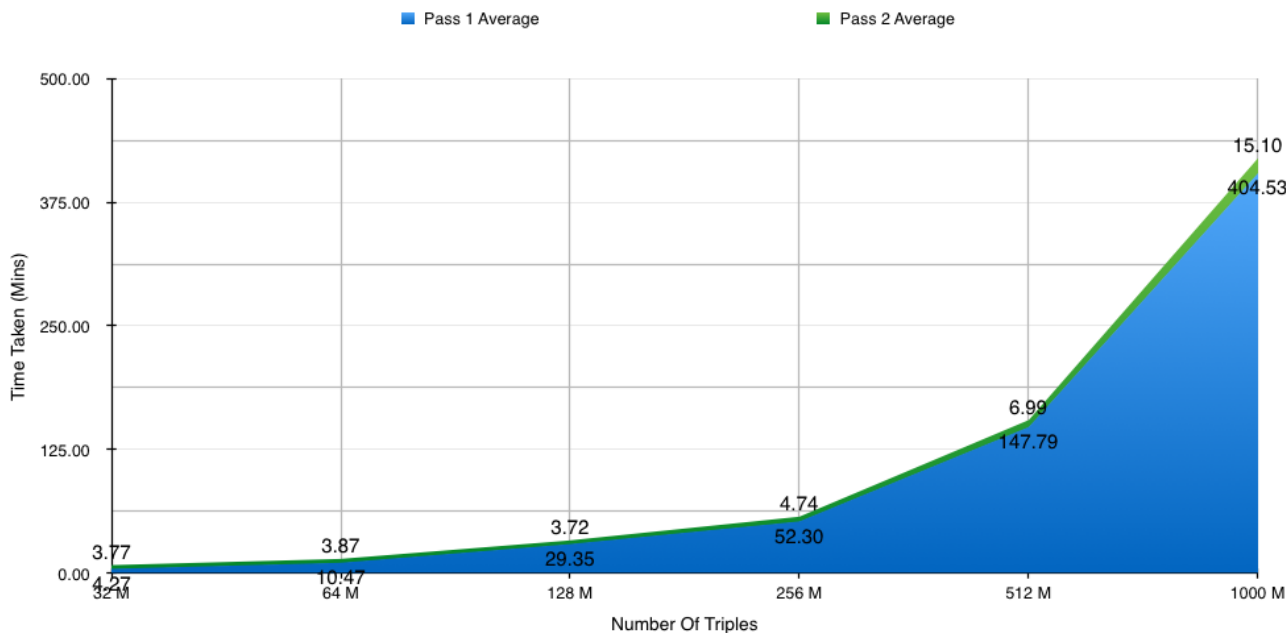


FIGURE 7.15: Approach Two - Query Performance Across 8 Nodes

7.8.2 Performance Scalability Across Number Of Nodes

Figure 7.16 shows how approach two scales across the range of cluster sizes. Only the eight node cluster was able to complete the queries on 1000M triples, with all other cluster sizes not having sufficient space on the HDFS. This is one drawback of approach two, as the input data is not compressed: the intermediate results require more space on the HDFS. One recurring theme displayed in Figure 7.16 is that an increase in node number always results in a corresponding decrease in running time. However, the amount by which run time is decreased is not consistent as the number of nodes is increased.

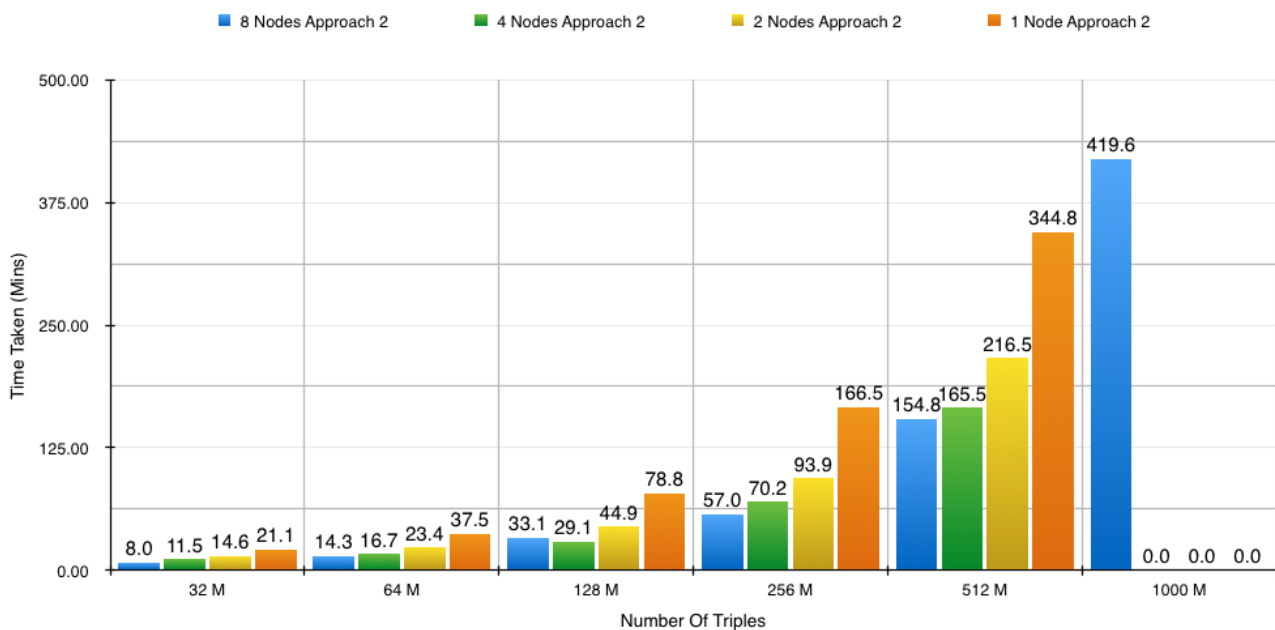


FIGURE 7.16: Approach Two - Query Performance Scalability Across Nodes

7.8.3 Speed-Up Analysis Of Approach Two's Query Stage

As with section 7.7.4, the speed-up of approach two's query time against cluster size was assessed. Again the speed-up was calculated by dividing the total time taken for the query algorithm of approach two across two, four and eight nodes against the time taken for one

node. Due to the increased network transfer required by approach two, the observed speed-up factor is further away from the theoretical max than the speed-up characteristic of approach one. However the speed-up factor for approach two is still always improved by adding additional nodes and increases towards the theoretical max as the input dataset size grows.

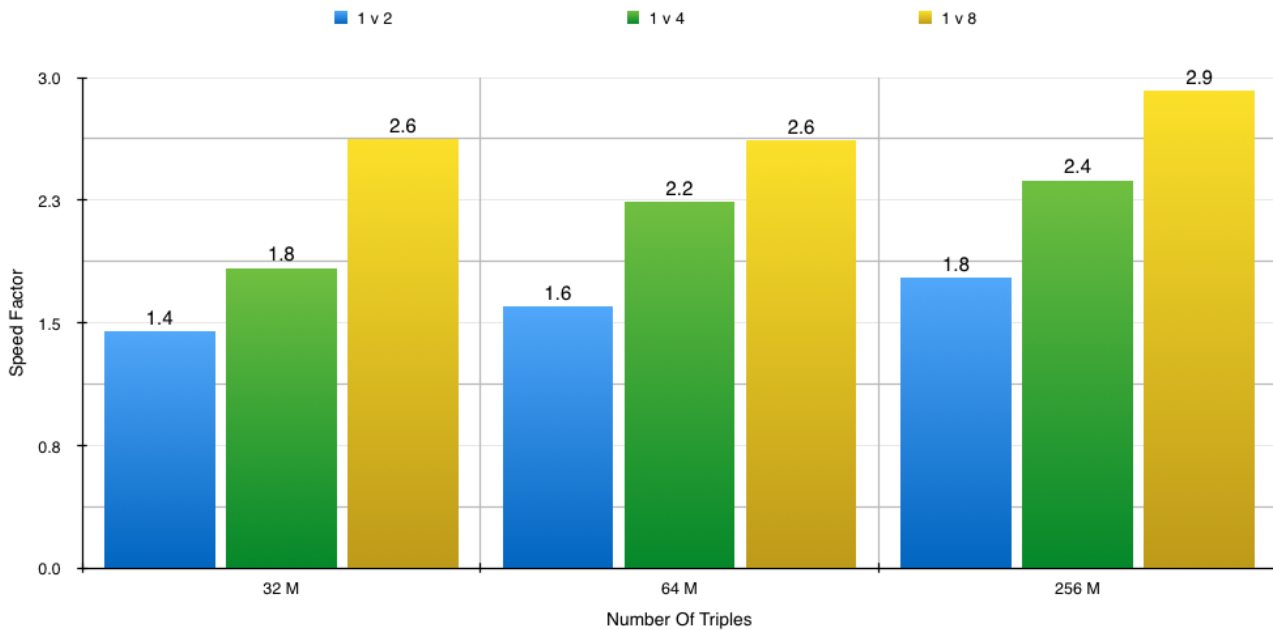


FIGURE 7.17: Approach Two - Query speed-up Factor

7.9 Cluster Approach Comparison

7.9.1 Approach Comparison On Eight Nodes

This section explores how the two approaches compare directly when running on the full eight node cluster. Figure 7.18 show a comparison between the query algorithm of approach one and approach two. The results show that approach two is consistently outperformed by approach one. This is expected and mirrors the results from the staff desktop machine.

However to more directly compare the two approaches, the combined upload and query time of approach one needed to be compared to approach two. This result, shown in Figure 7.19, shows that even with the additional time required to perform the upload algorithm,

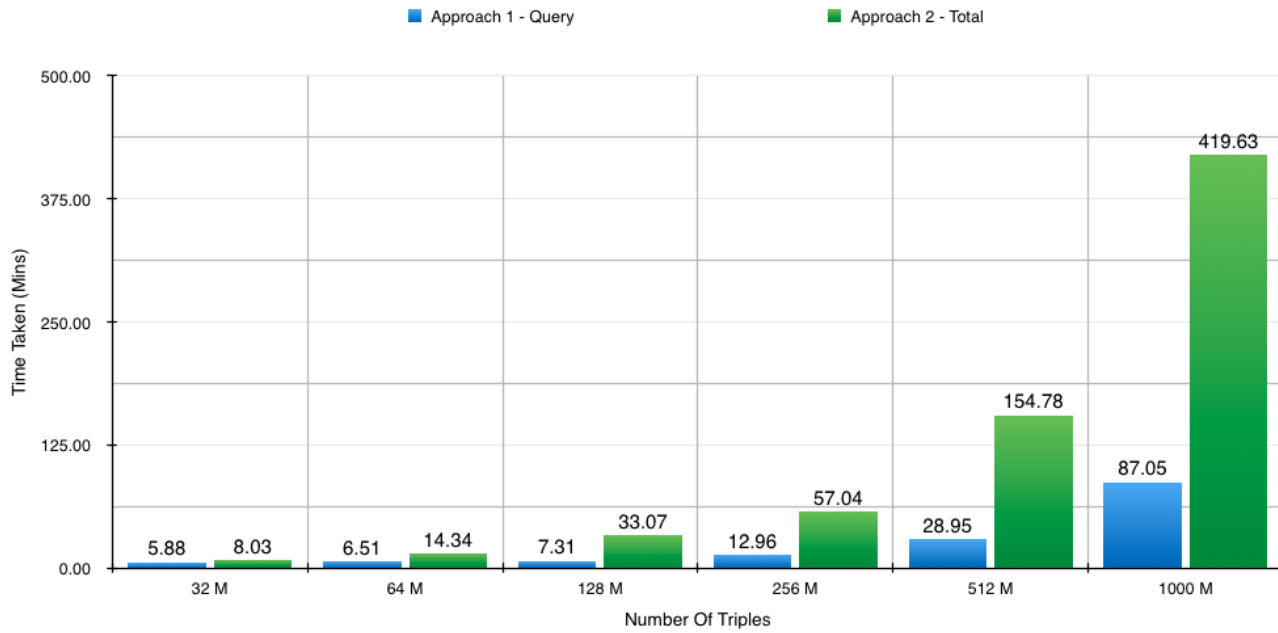


FIGURE 7.18: Eight Node Cluster Approach Query Comparison

approach one is quicker once the dataset size is increased past 128M. This contradicts the comparison found between the approaches when running on the single node, as discussed in section 7.5. This contradiction will be further explored in chapter 8.

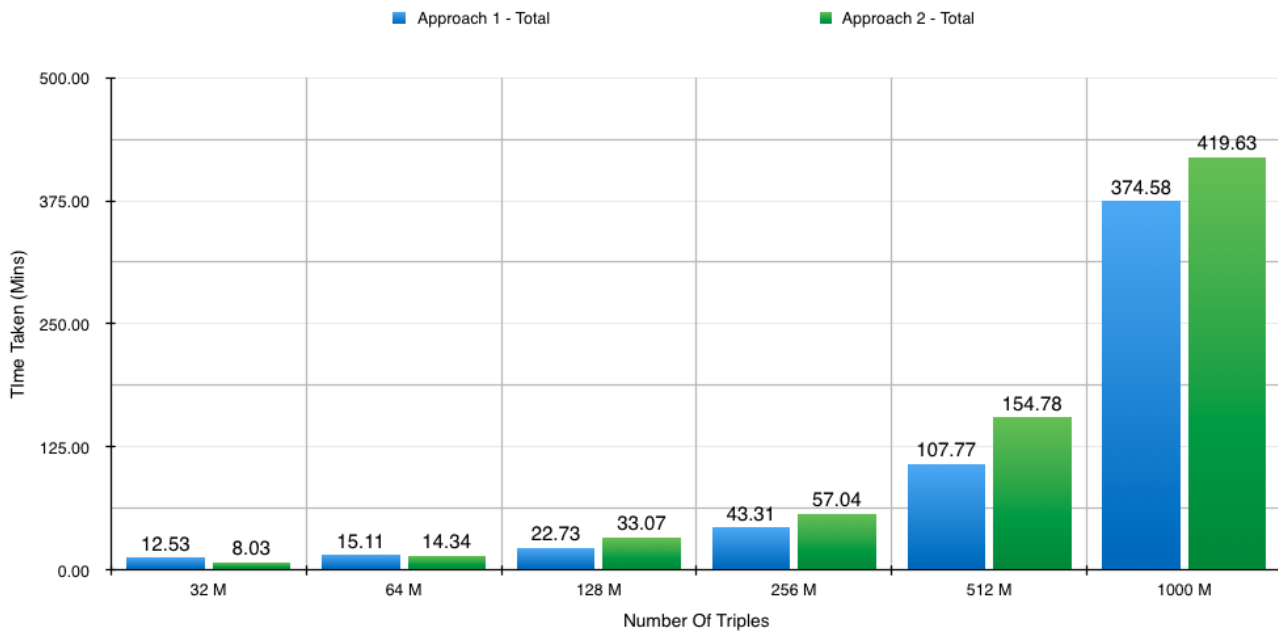


FIGURE 7.19: Eight Node Cluster Approach Total Comparison

7.9.2 Approach Comparison On Four Nodes

To see if the performance characteristics from comparing the two approaches on eight nodes are repeated when running on other cluster sizes, the approaches were compared when running on four nodes. The results can be seen in Figure 7.20. The results for 1000M on approach two are missing due to the lack of HDFS storage space on the four node cluster. The Figure shows that, when running on the four node cluster, approach two is always the better performing of the two stages across all dataset sizes. This contradicts the results from the eight node cluster where approach one was the best performing stage. This characteristic will be explored in greater depth in chapter 8.

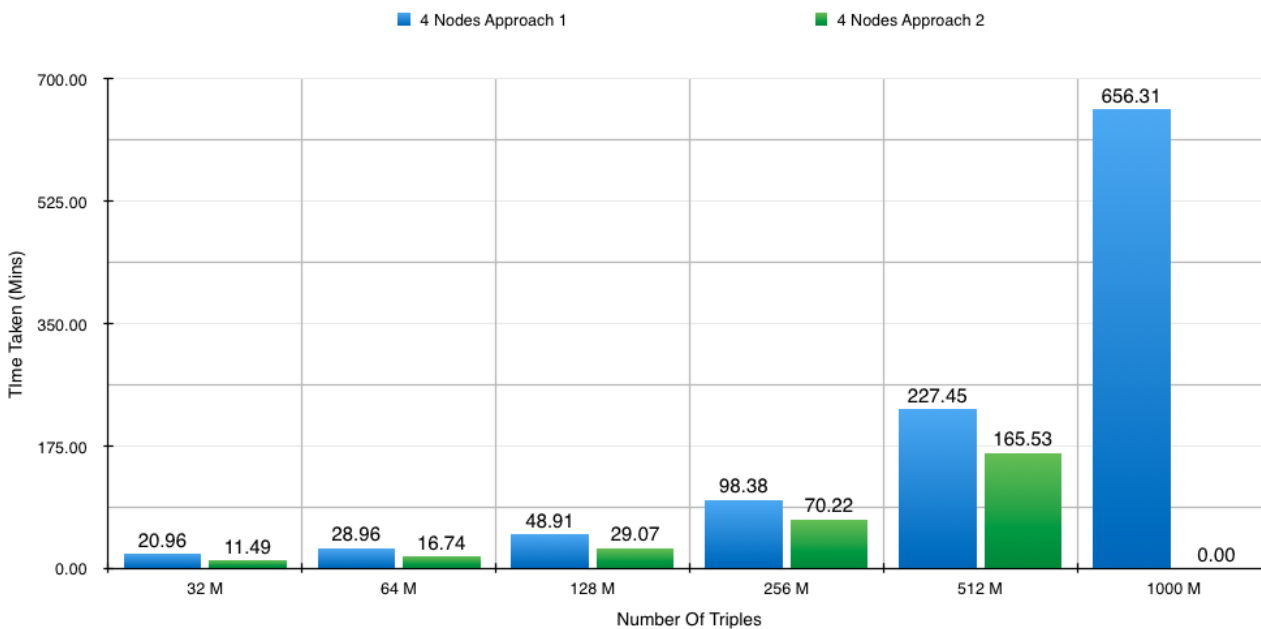


FIGURE 7.20: Four Node Cluster Approach Total Comparison

7.9.3 Approach Comparison With Jena

To test how the two approaches compare with the Jena TDB approach, Jena was run on one of the nodes which comprises the Hadoop cluster. This allowed for direct comparisons to be drawn. Figure 7.21 shows how the two approaches, running on the full eight node cluster, compare against Jena running on a single node. The dataset size of 128M triples was the maximum size which Jena was able to successfully run against. This shows how effective

both the Hadoop approaches are when compared with Jena, with Hadoop approach one being nearly 20 times faster than Jena.

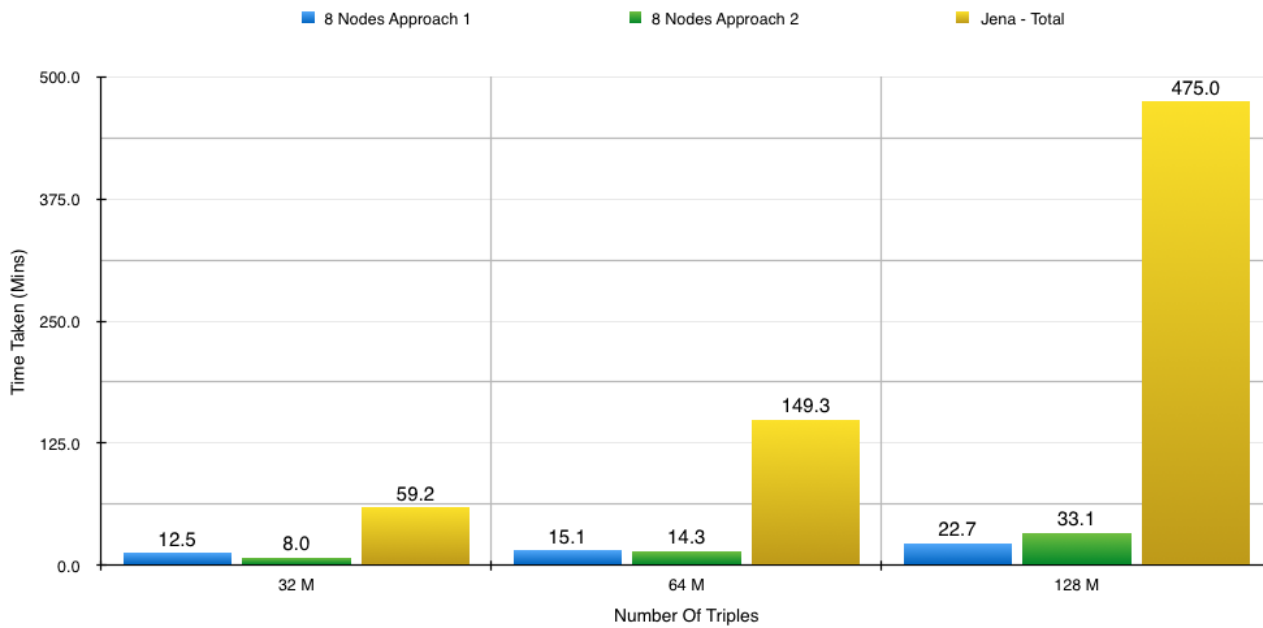


FIGURE 7.21: Hadoop Approaches Versus Jena

7.10 Naïve versus Optimised Implementation

The two Hadoop approaches were also tested against a naïve implementation. This naïve implementation processes the data in a similar manner to many of the existing Hadoop RDF systems and makes no use of map-side or broadcast joins. The naïve implementation must complete all the queries via a series of cascade reduce-side joins. However, it still creates a super-query to avoid the repetition of joins and makes use of the multi-join method to re-use the same reducers to join unrelated triples groups simultaneously. This naïve implementation requires two extra complete Map/Reduce iterations to join all the triples. Figure 7.22 shows how the naïve implementation compares with the two optimised approaches when running on an eight node cluster. It shows that the naïve implementation performs worst once the dataset size is increased past 32M triples. The performance deficit of the naïve implementation to the optimised approaches grows as dataset size increases. It is also

worth noting that the naïve implementation was unable to process dataset sizes larger than 128M triples.

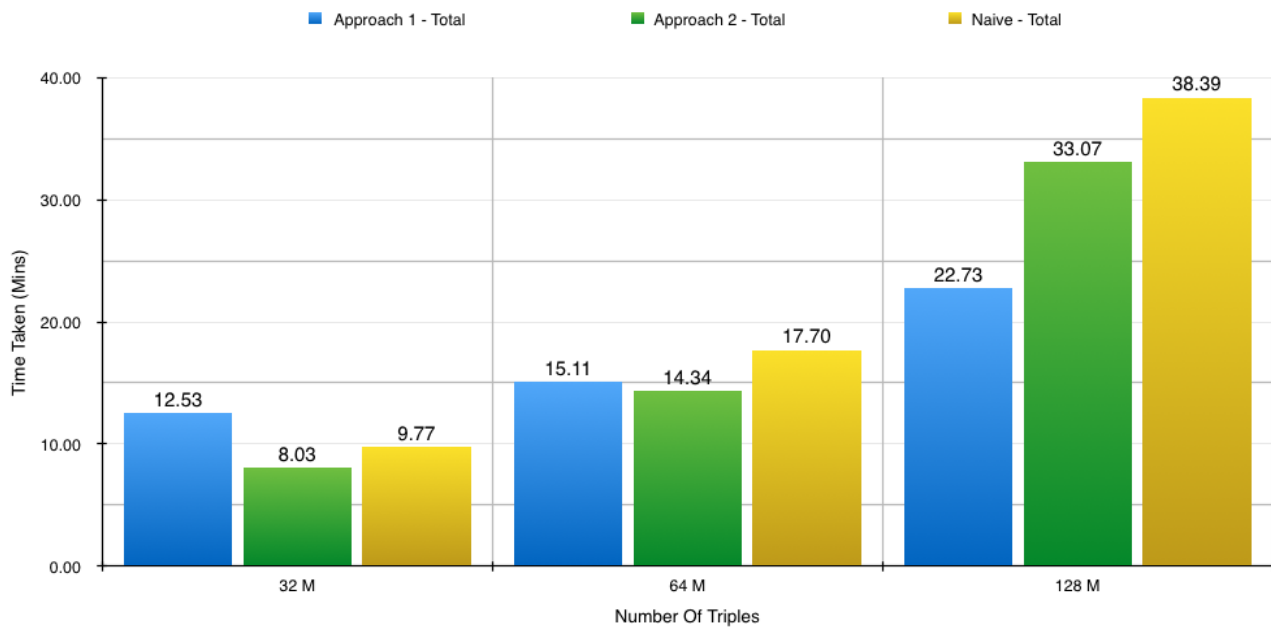


FIGURE 7.22: naïve versus Optimised Implementation

Chapter 8

Interpretation and Discussion Of Results

8.1 Summary and Significance Of Results

Overall the results show that both Hadoop-based approaches achieve the three objectives for the required software functionality, as detailed in section 5.4.1. These can be summarised as: returning the same response as the original SPARQL queries, demonstrating greater performance than Jena TDB and being scalable when running in a distributed environment.

8.1.1 Single Node Results

Section 7.5 highlights that even when running on a single machine, both the upload and query performance of Jena TDB was beaten by the Hadoop approaches. That Hadoop is better performing was somewhat unexpected, as Jena is designed for use on a single machine and Hadoop is not. The Hadoop approaches are designed to distribute the workload across a cluster and are therefore not optimised for single machine use. These results also highlight just how unsuitable Jena TDB is for processing NHS sized datasets.

As detailed in section 7.5, both approaches take an un-characteristically large amount of time to process 1000M triples. This performance trait is consistent across the upload and query stages of approach one and approach two. This suggests that this is the limit of the dataset size which can be processed by the two Hadoop approaches when running on the single machine. However this is still substantially more than Jena was able to process on the same hardware.

8.1.2 Cluster Results

The results demonstrate that both the Hadoop approaches scale well when run in a distributed environment. When run on various cluster sizes, both the approaches always demonstrate a decrease in run-time as cluster size increases. This highly scaleable nature of the two approaches is very encouraging and validates the reasons for choosing Hadoop as the basis for the new framework, which were established in section 5.4.2.

8.1.3 Cluster Size Versus Speed-Up Factor

Figure 7.14 shows the speed-up factor of approach one and Figure 7.17 the speed-up of approach two. These figures show that an increase in cluster size always results in a decrease in runtime for both approaches. However, there are two interesting trends that emerge from further analysis of the speed-up results.

It can be seen that approach one responds better to increases in node numbers, when compared to approach two. For example, going from a cluster size of one to eight will speed up query time for approach one by a factor of 5.9 with a dataset size of 256M triples. Whereas approach two only produces a speed-up factor of 2.9 for the same increase in nodes and dataset size. This shows that approach one makes more efficient use of the available cluster resources. This could be due to approach one's use of the map-side join technique. This technique allows for better use of the cluster, as each node can do more independently, without the need to transfer data over the network to a reduce task. Approach two uses the reduce-side method for all joins and so nodes must wait to send the data to

reduce tasks. This means that individual nodes can do less independently, explaining why approach two utilises extra nodes less efficiently when compared with approach one.

Another interesting characteristic is that the speed-up from a certain cluster size varies dependant on input dataset size. For example, approach one's speed-up going from one to eight nodes is only 2 at a dataset size of 32 million triples. However, the speed-up factor increases dramatically to 5.9 at 256M triples. This means that the more data that has to be queried by approach one, the more optimally the Hadoop cluster will utilise its available nodes. This trend is not only present going from just a size of one to eight, indeed the speed-up for all cluster sizes increases towards the optimal value as dataset size increases. This is clearly a very promising trait when dealing with NHS sized datasets. While this characteristic is present in the speed-up result for approach two, it is not as pronounced. Again, this could be due to the increased network transfers which the approach two algorithm requires.

Further tests on greater cluster sizes would need to be performed to assess at what point an increase in nodes does not result in a decrease in run time. Research conducted by Fadika, Dede, Govindaraju, and Ramakrishnan (2011) show that Hadoop's performance increase does diminish once the cluster is increased past a certain size. However this is dependant upon the specific task being run.

8.1.4 Approach Comparison

Figure 7.8 shows the comparison between the total time taken for the two approaches on the single machine. When just considering the two query stages between the two approaches, approach one is substantially faster than two. However, once the upload stage is included with the query stage, approach two is the better performing approach. The faster query stage of approach one would suggest it is more suited to a scenario where the data will be queried multiple times and thus needs to be stored for longer. Then the performance cost of the upload stage would be offset by the performance benefit of approach one's query stage. Approach two would be more suited to a scenario where the data only needed to be queried once and the errors removed from the dataset. The performance difference between the approaches is interesting when considering the results from the single machine, as one

of the biggest performance bottle necks for Hadoop, transferring data over the network, is not a factor. This allows for a more direct and accurate comparison between approaches. As expected, approach one uses the pre-processed data to increase query performance relative to approach two. Even though the use of the map-side join is less of a benefit when running on a single machine, approach one is still performing its conditional logic on and writing to disk smaller triple elements due to the compressed data. The reduce task of approach one's selection stage is also doing less work than that of approach two, due to the pre-formatted data. These two reasons are the main factors behind approach one's superior query stage when running on the single machine.

The comparison between approaches is not as clear cut when considering the cluster results. From a cluster size of one to four nodes, the results mirror those from the single machine, with approach two having better performance when compared with the total time taken for approach one. However, as Figure 7.19 shows, once the cluster size is increased to eight nodes, approach one becomes the better performing approach, even with its costly upload stage included. The disparity between the single, two and four node cluster against the eight node cluster must be down to the increased demand in the network which an eight node cluster requires.

Overall, the results suggest that approach two is more suited to running on smaller cluster sizes due to its performance being affected by the additional network transfer that larger cluster sizes incur. They also suggest that approach one appears to be more scalable and is therefore suited to larger cluster sizes. This is because each node can do more work independently and requires less internode communication. On cluster sizes of four and below, approach two is more suitable for situations where the data only needs to be queried once. Approach one is more suitable if multiple queries have to be run, as the cost of the upload stage would quickly be nullified after only a few queries. However further tests on much larger clusters could be performed to assess if these performance characteristics continue.

8.1.5 Underlying Pass Comparison

From reviewing all the results, one clear pattern emerges regarding the underlying passes which comprise the approaches. Any stage which has to search the input datasets is always the worst performing stage. In the upload algorithm, this is the sort-on subject stage and in the query algorithms, this is the selection stage. Both these stages have to traverse the entire input datasets to search for the required BGP to complete the queries. Traditional databases use indexes of the data to directly access specific records. Hadoop's lack of indexes means that the entire input dataset must be scanned through to select required records. This explains why the selection stages of the approaches are the slowest stages. This reinforces research by Kulkarni (2010) that Hadoop's lack of data filtering, leading to the whole input being scanned, can have a serious impact on triple selection.

The join stage appears to be highly optimal and demonstrates impressive performance, even when joining one billion triples. Figure 7.11 displays the results for approach one on the eight node cluster. It shows that one billion triples are successfully joined in under eight minutes. This seems to contradict the literature discussed in section 3.3, which highlights joins as being one of the most complex and costly Map/Reduce operations. That the join stages are the best performing of the two stages, shows how optimised joining via a broadcast join, over a series of cascade reduce-side joins, makes the algorithms.

8.2 Comparison With Literature

Section 7.10 demonstrates how the two Hadoop-based approaches compare when run against a naïve implementation. Comparing the optimised to the naïve implementation shows how effective the use of the broadcast joins, over cascade reduce-side joins, really is. The optimised approach not only enables greater performance but also allows larger dataset sizes to be processed. Two key advantages which show that, wherever possible, the broadcast join should be used instead of the cascade reduce-side join which is used in the existing literature. However, the naïve implementation used as the comparison was

not truly naïve as it still employed a selection of optimisation techniques. A truly naïve implementation, based on the currently available literature, would not re-use reduce tasks to join unrelated triple groups. Instead, these triple groups would each have to be joined via their own complete Map/Reduce iterations. As none of the current solutions perform any analysis of the SPARQL queries to be run before completing the joins, they would all re-compute the common joins. This would have a massive impact on performance, as the total time taken would increase proportionately, depending on the total number of queries to be run. This would suggest that a truly naïve implementation would demonstrate even worse performance when compared with the optimised approaches created for this project.

Comparisons between the results presented here and those from the currently available literature are possible. However variations between test environment, datasets and SPARQL queries make any conclusions drawn from the comparisons hard to make. Section 3.6.2 discusses the presented results from the current literature. The best performing of the existing solutions is the clique-square implementation presented by Goasdoué et al. (2013). This work is also the only solution to use the map-side join technique and thus is the most comparable to approaches created for this project. For a dataset size of 1000M triples the clique-square approach took a total time of 316 minutes to both upload and run a single moderately complex query. Table 8.1 shows the specification of the cluster on which the performance of the clique-square approach was assessed. As can be seen, the clique-square framework was tested on a cluster of much greater performance than the cluster used for this project.

Component	Node Specification
CPU	Intel Xeon 2.93GHz Quad Core
RAM	16GB
HDD	2 x 600GB RAID1

TABLE 8.1: Specification Of The Clique-Square Hadoop Cluster

Section 7.9 shows total time taken for the two approaches created for this project to process 1000M triples on the eight node cluster. This was 374 minutes for approach one and 419 minutes for approach two. Comparisons between these times and the time taken for clique-square are hard to make, as clique-square only runs one query, whereas the two Hadoop

approaches are completing eight. However, one method for allowing a more direct comparison would be to multiply the query portion of clique-squares total time by eight. This would reflect the total number of queries that the approaches created for this project complete. The query portion comprises 59 of the total 316 minutes for the clique-square approach. Using this method would create a new total run time of 729 minutes for the clique-square approach to perform the same number of queries as the two approaches created for this project. While it would be hard to draw significant conclusions from using this method, it does suggest that the two Hadoop-Medical-RDF approaches have better performance than clique-square, even when the results for the clique-square approach were gathered from a cluster of more modern and capable machines.

This hypothetical result could be used to show how effective the unique techniques employed for this project are in increased query performance. As the clique-square approach uses map-side joins, its performance deficit to the newly created approaches could be due to the creation of the super-query (which eliminates common joins) and the use of the broadcast join method (which eliminates numerous Map/Reduce iterations).

Chapter 9

Conclusions And Further Work

9.1 Conclusions

9.1.1 Project Summary

This thesis has attempted to provide a solution to the problem of extremely slow query and upload performance of an existing Semantic Web method used to assess the quality of medical RDF data. Hadoop was chosen as the basis for the project as existing literature shows it to be an effective way of parallelising a big data problem across a cluster, leading to a fast and scalable solution. Two separate approaches to solving the problem were created, each using alternative Hadoop joining strategies. Both of the approaches meet the required software functionality established in section 5.4.1. Each approach has its own set of strengths and weaknesses, leading to them having their own distinct scenario in which each would be best suited. Approach one is more suited to running on larger cluster sizes, as it scales well and is the better performing of the two once the number of nodes is increased past eight. It is also ideal for situations where the data will be queried more than once as the upload stage allows for extremely fast querying. Approach two is more suited to smaller cluster sizes and for when the data only needs to be queried once. The success or otherwise of the approaches was determined by benchmarking them against the existing Semantic Web solution and against a naïve Hadoop implementation.

The work conducted for this paper also resulted in a publication being submitted and is currently waiting for review (Bonner, Antoniou, Moss, & Kureshi, 2014). The paper focused on whether Hadoop is a viable platform to assess the quality of medical data and compares the performance of approach one and comparisons with Jena. The paper was submitted to the ASE Big Data Science International Conference 2014 to be held in Beijing.

9.1.2 Aims and Objectives Achieved

All of the aims and objectives for this project, specified in section 1.3, have been successfully achieved.

- i) Firstly, two alternative frameworks were created in Apache Hadoop to both store and query medical RDF data.
- ii) Secondly, the newly created approaches were successfully tested on varying quantities of medical data.
- iii) Finally the two Hadoop approaches were tested against the existing Jena implementation and against a naïve implementation based on the existing literature.

9.1.3 Evaluative Conclusions

Throughout the course of this project, the following points have become apparent:

- i) From such positive results it can be said that Hadoop is very effective at storing and querying medical RDF data, when compared with Jena. Both the Hadoop approaches demonstrate better performance than Jena when running on the same machine and processing the same datasets. The Hadoop approaches also demonstrate good scalability when tested in a distributed environment, meaning that they are well equipped to deal with NHS-sized datasets.
- ii) The project uses the broadcast join method for completing many of the required joins. This method reduces the need for a series of cascade reduce-side joins. This project

appears to be the first from the currently available literature to make use of the broadcast join method to perform queries on RDF data.

- iii) The project reinforces the idea, introduced by White (2010), that the map-side join method is much more efficient than the reduce-side method. This can be seen in how much quicker approach one's query stage is when compared to approach two's. This difference is solely down to the use of map-side joins. The use of map-side joins is particularly suited to larger cluster sizes as it dramatically reduces the volume of network traffic.
- iv) This project also appears to be the first to introduce the notion of the super-query. A super-query is created from a series of standard SPARQL queries and is used to save on the re-computation of common joins. This is something neither Jena nor the existing Hadoop-SPARQL approaches appear to implement.
- v) One clear non-performance related advantage to using Hadoop to query medical data, rather than a dedicated triplestore cluster like the clustered Jena system introduced by Owens et al. (2008), is that a Hadoop cluster is inherently multi-functional. Creating a triplestore cluster would limit the machines used to just the task of processing RDF data. Using Hadoop to process the medical data would also allow the cluster to be used for other tasks, hopefully improving the total utilisation of the resource. This could make the setup and running cost of a Hadoop cluster easier to justify. Using Hadoop as the basis for the framework enables greater portability, as companies such as Amazon already offer Hadoop as a service, on demand in the Cloud (Kambatla, Pathak, & Pucha, 2009).
- vi) Using Hadoop as the basis of the framework allows for much greater functionality over what is offered by traditional SPARQL queries. For example both the approaches utilise the Hadoop counter feature to allow the user to check the exact quantity of records which failed a specific error check. This summation would have to be performed manually using the Jena approach. The Hadoop approaches could easily be expanded to remove the false records from the original input data, leaving an error-free dataset for use in further research. Again, this is functionality that is impossible to replicate using SPARQL alone.
- vii) The two Hadoop approaches have been designed to enable expansion beyond the current queries. Any future users of the framework could add additional queries with

minimal effort and without having a detrimental effect on performance. However the time taken to add an additional query to the approaches is still significant, especially when compared to the time taken to write a SPARQL query.

- viii) Hadoop could potentially have lots of other uses within the medical data domain, in addition to being used for error checking. The results gained from this project can be used as evidence that Hadoop should be strongly considered for future use when large scale medical data needs processing.
- ix) In addition to using Hadoop to query NHS scale RDF datasets, some of the key optimisations used in this project could easily be expanded for use with other queries and datasets. This project shows that scanning the input datasets to see which triples groups are suitable for joining via the broadcast method is a key way to reduce query time. Another optimisation is the use of a pre-processing stage to enable the use of the highly efficient map-side join method in the triple selection stages.

9.2 Further Work

9.2.1 Improvements To The Project

With hindsight, there are several ways in which this project and the testing of it could have been improved. These are mostly focused on the test environment and test datasets used.

Due to privacy issues relating to sensitive medical data, it was only possible to test the frameworks on synthetically generated datasets. While this generated data closely mimics the structure and distribution of the real-world data, it would be good to test that the frameworks actually perform as well on real-world data.

Due to time and resource restraints, the framework was only tested up to a total dataset size of one billion triples. One way of further testing the project would be to use massive dataset sizes of over one billion triples. Based on the current results it is possible to speculate how the frameworks would behave on a dataset size of two billion triples. Running on the standard desktop machine, both the frameworks would struggle to process two billion triples

in a reasonable time. Running on the eight node cluster, it would be possible to suggest that approach one could cope well with two billion triples. Both frameworks show that they scale well as cluster size is increased. To further test the project, the frameworks could be run on much larger clusters to assess whether and how the scalability characteristics continue.

9.2.2 Expansion Of The Project

As a result of completing this project, it can be seen that there are evidently ways in which it could be expanded and lead to future research. Recently the latest version of Hadoop, version 2, has been released. This version replaces the Map/Reduce framework with YARN, a substantially different way of implementing the core Hadoop concepts. Full details are provided by Vavilapalli et al. (2013). While this new version of Hadoop was released too late for it to form the basis of this project, future research could be performed to port the code into this new framework. Any performance differences could also be explored to see if this new version can bring speed-up benefits.

As the project results show, the stage of any of the developed algorithms which consumes the greatest amount of time, is the stage which has to traverse the original input dataset. For the two Hadoop query algorithms, this is the selection stage and for the upload algorithm it is the sort-on subject stage. This problem of traversing the input data is inherent to the Map/Reduce framework, since it can only have a complete file or collection of files as input. It has no method of targeting specific portions of a file and thus must scan through the complete file to locate it. This task of locating items is usually handled in traditional databases by an index, but none currently exist for Hadoop. Due to the distributed nature of Hadoop, creating an index could prove difficult, as it would not only have to track records location within files, but also the machine on which it is located. Future research could explore the creation of an index for Hadoop. There has been some early work conducted in this direction, for example work by Liao, Han, and Fang (2010), although there is still massive scope for future research.

Currently this project is customised to process medical RDF data and specific SPARQL queries. However future research could be performed to allow the framework to process

any datasets and queries. Research would also need to be done to allow the optimisation developed for this project to be performed automatically. Specifically this means, researching a way of automatically traversing the datasets to see the distribution of the triple groups and discovering which would be of a suitable size for joining via a broadcast join. Research could also be performed to allow for automatic creation of a super query from a subset of queries.

All of the values taken for the cluster results were gathered from a dedicated Hadoop cluster. It is possible that future users of the developed frameworks might not have access to a dedicated resource. Further research could be done to investigate how the frameworks would cope when moved from a dedicated resource to a Cloud-based resource such as Amazon's EC2 Map/Reduce offering. Research has been performed by many people including Kambatla et al. (2009) to show that Hadoop can successfully be moved to the cloud, however using the cloud might not be suitable for this project. This is due to the large privacy concerns that using a publicly available Cloud service to store and process confidential medical data entails. However, the potential performance benefit and cost saving created by being able to spawn a Hadoop cluster dynamically, depending on the volume of data to be processed, is definitely worthy of further investigation.

9.3 Final Conclusions

This project can be considered to have been successful, as two alternative Hadoop-based approaches to assess the quality of medical data have been created. The positive results gained from comparative testing between the two approaches and Jena can be used to provide a definitive conclusion to the project. Hadoop is a more effective method to both store and also query massive RDF datasets than the existing Semantic Web solution. Hadoop allows for a system to assess the quality of medical data which is not only better performing than using traditional Semantic Web technologies, but it is also able to process the massive volumes of data required by the NHS, as it scales the workload across a computer cluster. This project has also introduced two novel methods for use when completing SPARQL queries using map/reduce: the use of broadcast joins and the creation of the super-query.

The work performed for this project will hopefully be used to allow for more and larger medical databases to be checked for errors and inconsistencies. This should hopefully lead to more accurate and reliable databases being used in both medical research and also for diagnosis.

References

- Afrati, F. N., & Ullman, J. D. (2010). Optimizing joins in a map-reduce environment. In *Proceedings of the 13th international conference on extending database technology* (pp. 99–110). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/1739041.1739056> doi: 10.1145/1739041.1739056
- Agrawal, D., Bernstein, P., Bertino, E., Davidson, S., Dayal, U., Franklin, M., ... Widom, J. (2011). Challenges and Opportunities with Big Data 2011-1.
- Antoniou, G., & Harmelen, F. v. (2008). *A semantic web primer, 2nd edition (cooperative information systems)* (2nd ed.). The MIT Press.
- Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiawicz, J., ... Wawrzynek, J. (2009). A view of the parallel computing landscape. *Communications of the ACM*, 52(10), 56–67.
- Berners-Lee, T., Hendler, J., & Lassila, O. (2001, May). The semantic web. *Scientific American*, 284(5), 34-43. Retrieved from <http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21>
- Bizer, C., Heath, T., & Berners-Lee, T. (2009). Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3), 1–22.
- Blanas, S., Patel, J. M., Ercegovac, V., Rao, J., Shekita, E. J., & Tian, Y. (2010). A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 acm sigmod international conference on management of data* (pp. 975–986). New York, NY, USA: ACM.

- Bonner, S., Antoniou, G., Moss, L., & Kureshi, I. (2014). Using hadoop to implement a semantic method of assessing the quality of research medical datasets. *ASE Big Data Science*.
- Chambers, I., Gregson, B., Citerio, G., Enblad, P., Howells, T., Kiening, K., ... Ragauskas, A. (2009). Brainit collaborative network: analyses from a high time-resolution dataset of head injured patients. In *Acta neurochirurgica supplements* (pp. 223–227). Springer.
- Chandar, J. (2010). *Join algorithms using map/reduce* (Unpublished master's thesis). University of Edinburgh.
- Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... Gruber, R. E. (2006). Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th unix symposium on operating systems design and implementation - volume 7* (pp. 15–15). Berkeley, CA, USA: USENIX Association. Retrieved from <http://dl.acm.org/citation.cfm?id=1267308.1267323>
- Corsar, D., Moss, L., & Piper, I. (2012). Data quality assessment using linked data: A case study in the medical domain. In *E-kaw 2012, the 18th international conference on knowledge engineering and knowledge management*. Retrieved from [http://ekaw2012.ekaw.org/sites/ekaw2012.ekaw.org/files/ekaw2012pdsbmission21\(2\).pdf](http://ekaw2012.ekaw.org/sites/ekaw2012.ekaw.org/files/ekaw2012pdsbmission21(2).pdf)
- Davies, J., Fensel, D., & van Harmelen, F. (Eds.). (2003). *Towards the semantic web: Ontology-driven knowledge management*. Chichester, UK: Wiley.
- Dean, J., & Ghemawat, S. (2004). Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on symposium on opearting systems design & implementation - volume 6* (pp. 10–10). Berkeley, CA, USA: USENIX Association. Retrieved from <http://dl.acm.org/citation.cfm?id=1251254.1251264>
- DuCharme, B. (2011). *Learning sparql* (S. St. Laurent & J. Perez, Eds.).
- Fadika, Z., Dede, E., Govindaraju, M., & Ramakrishnan, L. (2011). Benchmarking mapreduce implementations for application usage scenarios. In S. Jha, N. gentschen Felde, R. Buyya, & G. Fedak (Eds.), *Grid* (p. 90-97). IEEE. Retrieved from <http://dblp.uni-trier.de/db/conf/grid/grid2011.html#FadikaDGR11a>

- Feldman, B., Martin, E. M., & Skotnes, T. (2012). Big data in healthcare hype and hope.
- Gartner. (2011). Pattern-based strategy: Getting value from big data. Retrieved from <http://www.gartner.com/it/page.jsp?id=1731916>
- Goasdoué, F., Kaoudi, Z., Manolescu, I., Quiané-Ruiz, J., & Zampetakis, S. (2013). CliqueSquare: efficient Hadoop-based RDF query processing. In *BDA'13 - Journées de Bases de Données Avancées*. Nantes, France. Retrieved from <http://hal.inria.fr/hal-00867728>
- Goldberg, S. I., Niemierko, A., & Turchin, A. (2008). Analysis of data errors in clinical research databases. In *Amia annual symposium proceedings* (Vol. 2008, p. 242).
- Guo, Y., Pan, Z., & Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3). Retrieved from <http://www.websemanticsjournal.org/index.php/ps/article/view/70>
- Harris, S., Lamb, N., & Shadbolt, N. (2009). 4store: The design and implementation of a clustered rdf store. In *5th international workshop on scalable semantic web knowledge base systems (ssws2009)* (pp. 94–109).
- Haslhofer, B., Roochi, E. M., Schandl, B., & Zander, S. (2011, March). *Europeana rdf store report* (Technical Report). Vienna: University of Vienna. Retrieved from <http://eprints.cs.univie.ac.at/2833/>
- Howe, D., Costanzo, M., Fey, P., Gojobori, T., Hannick, L., Hide, W., ... others (2008). Big data: The future of biocuration. *Nature*, 455(7209), 47–50.
- Husain. (2009). *Cost-based query processing for large rdf graph using hadoop and mapreduce* (Tech. Rep.).
- Husain, Doshi, P., Khan, L., & McGlothlin, J. (2009). *Efficient query processing for large rdf graphs using hadoop and mapreduce* (Tech. Rep.).
- Husain, McGlothlin, J. P., Masud, M. M., Khan, L. R., & Thuraisingham, B. M. (2011). Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Trans. Knowl. Data Eng.*, 23(9), 1312-1327. Retrieved from <http://dblp.uni-trier.de/db/journals/tkde/tkde23.html#HusainMMKT11>

- Joshi, S. B. (2012). Apache hadoop performance-tuning methodologies and best practices. In *Proceedings of the 3rd acm/spec international conference on performance engineering* (pp. 241–242). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/2188286.2188323> doi: 10.1145/2188286.2188323
- Kambatla, K., Pathak, A., & Pucha, H. (2009). Towards optimizing hadoop provisioning in the cloud. In *Proc. of the first workshop on hot topics in cloud computing* (p. 118).
- Kulkarni, P. (2010). *Distributed sparql query engine using mapreduce* (Unpublished master's thesis). University of Edinburgh.
- Lam, C. (2010). *Hadoop in action*. Manning Publications.
- Liao, H., Han, J., & Fang, J. (2010). Multi-dimensional index on hadoop distributed file system. In *Networking, architecture and storage (nas), 2010 ieee fifth international conference on* (pp. 240–249).
- Lin, J., & Dyer, C. (2010). *Data-intensive text processing with mapreduce*. Morgan and Claypool Publishers. Retrieved from <http://dx.doi.org/10.2200/S00274ED1V01Y201006HLT007>
- Mazumdar, S. (2011). *Complex sparql query engine for hadoop mapreduce* (Unpublished master's thesis). University of Sophia Antipolis.
- Miner, D., & Shook, A. (2012). *Mapreduce design patterns building effective algorithms and analytics for hadoop and other systems*. Oreilly & Associates Inc.
- Moss, L., Corsar, D., & Piper, I. (2012). A linked data approach to assessing medical data. In P. Soda & F. Tortorella (Eds.), *25th international symposium on computer-based medical systems (cbms), 2012* (pp. 1–4). doi: 10.1109/CBMS.2012.6266391
- Myung, J. (2010). *Sparql processing with mapreduce*. Retrieved from <http://ids.snu.ac.kr/w/images/a/a7/2010SS-11.ppt>
- Myung, J., Yeon, J., & Lee, S.-g. (2010). Sparql basic graph pattern processing with iterative mapreduce. In *Proceedings of the 2010 workshop on massive data analytics on the cloud* (pp. 6:1–6:6). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/1779599.1779605> doi: 10.1145/1779599.1779605

- Ogbuji, U. (2000). *An introduction to rdf*. Retrieved 2000, from <http://www.ibm.com/developerworks/library/w-rdf/>
- Owens, A., Seaborne, A., & Gibbins, N. (2008). *Clustered TDB: A clustered triple store for jena* (Tech. Rep.). Electronics and Computer Science, University of Southampton.
- Palla, K. (2009). *A comparative analysis of join algorithms using the hadoop map/reduce framework* (Unpublished master's thesis). University of Edinburgh.
- Patchigolla, V. (2011). Comparison of clustered rdf data stores.
- Rajaraman, A., & Ullman, J. D. (2012). *Mining of massive datasets*. Cambridge: Cambridge University Press. Retrieved from http://www.amazon.de/Mining-Massive-Datasets-Anand-Rajaraman/dp/1107015359/ref=sr_1_1?ie=UTF8&qid=1350890245&sr=8-1
- Rohloff, K., & Schantz, R. E. (2010). High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In E. Tilevich & P. Eugster (Eds.), *Psi eta*. ACM. Retrieved from <http://dblp.uni-trier.de/db/conf/oopsla/psieta2010.html#RohloffS10>
- Salati, M., Brunelli, A., Dahan, M., Rocco, G., Van Raemdonck, D., Varela, G., et al. (2011). Task-independent metrics to assess the data quality of medical registries using the european society of thoracic surgeons (ests) database. *European journal of cardio-thoracic surgery: official journal of the European Association for Cardio-thoracic Surgery*, 40(1), 91.
- Schmidt, M., Hornung, T., Lausen, G., & Pinkel, C. (2008). Sp2bench: A sparql performance benchmark. *CoRR*, abs/0806.4627.
- Segaran, T., Evans, C., & Taylor, J. (2009). *Programming the semantic web*. O'Reilly Media.
- Sequenda, J. (2013). *Introduction to: Triplestores*. Retrieved from http://semanticweb.com/introduction-to-triplestores_b34996
- Silberschatz, A., Korth, H., & Sudarshan, S. (2011). *Database systems concepts* (6th ed.). New York, NY, USA: McGraw-Hill, Inc.

- Soule, N. (2011). Efficient sparql query processing via map-reduce-merge.
- Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., & Reynolds, D. (2008). Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on world wide web* (pp. 595–604). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/1367497.1367578> doi: 10.1145/1367497.1367578
- Taylor, R. C. (2010). An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. *BMC Bioinformatics*, 11(S-12), S1. Retrieved from <http://dblp.uni-trier.de/db/journals/bmcbi/bmcbi11S.html#Taylor10>
- Urbani, J., Maassen, J., Drost, N., Seinstra, F. J., & Bal, H. E. (2013). Scalable rdf data compression with mapreduce. *Concurrency and Computation: Practice and Experience*, 25(1), 24-39.
- Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., ... others (2013). Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual symposium on cloud computing* (p. 5).
- W3C. (2001). *Semantic web architecture*. Retrieved 07-09-2013, from <http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html>
- W3C. (2004). *Rdf primer*. Retrieved 07-09-2013, from <http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>
- W3C. (2013). *Sparql 1.1 query language*. Retrieved 07-09-2013, from <http://www.w3.org/TR/sparql11-query/#BasicGraphPatterns>
- Weaver, J., & Williams, G. T. (2009). Scalable rdf query processing on clusters and supercomputers. In *The 5th international workshop on scalable semantic web knowledge base systems (ssws2009)* (p. 68).
- White, T. (2010). *Hadoop: The definitive guide* (Second Edition ed.; M. Loukides, Ed.). O'Reilly. Retrieved from <http://oreilly.com/catalog/9780596521981>
- Wilkinson, K., Sayers, C., Kuno, H., & Reynolds, D. (2003). Efficient RDF storage and retrieval in Jena2. In *Proc. first international workshop on semantic web and databases*. Retrieved from http://www.cs.uic.edu/~ifc/SWDB/papers/Wilkinson_etal.pdf

Appendix A

Appendix - SPARQL Queries

A.1 Original SPARQL Queries

```
prefix med: <http://www.abdn.ac.uk/~csc316/ontologies/ICUNeuro.owl#>
prefix pd: <http://www.abdn.ac.uk/~csc316/ontologies/PatientData#>
prefix mo: <http://www.abdn.ac.uk/~csc316/ontologies/MedicalObservations#>
prefix ann: <http://www.abdn.ac.uk/~csc316/ontologies/Annotations#>
prefix ms: <http://www.abdn.ac.uk/~csc316/ontologies/MedicalSensors#>
prefix oc: <http://www.abdn.ac.uk/~csc316/ontologies/ObservationCollection#>
prefix spec: <http://www.abdn.ac.uk/~csc316/ontologies/Specification#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn/>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
// check if a value is above the maximum acceptable value
SELECT ?obs ?p ?htime ?max ?value WHERE {
```

```
?range a med:AcceptableRange .
?range med:clinicalRangeMax ?max .
?range pd:hasParameter ?p .
```

```
?obs a mo:PhysiologicalObservation;
?obs ssn:observedProperty ?p .
?obs ssn:observationResultTime ?time .
?obs pd:atHumanTime ?htime .
?obs ssn:observationResult ?a1
```

```
?a1 ssn:hasValue ?a2 .
?a2 pd:readingValue ?value .
```



```
FILTER (?value > ?max)
}
```

```
---query.belowMinAcceptable
// check if a value is below the minimum acceptable value
SELECT ?obs ?p ?htime ?min ?value WHERE {
  ?range a med:AcceptableRange; med:clinicalRangeMin ?min; pd:hasParameter ?p.
  ?obs a mo:PhysiologicalObservation;
  ssn:observedProperty ?p; ssn:observationResultTime ?time; pd:atHumanTime ?htime.
  ?obs ssn:observationResult/ssn:hasValue/pd:readingValue ?value.
  FILTER (?value < ?min)
}
```

```
---query.belowMinAcceptablePlusSensorAccuracy2
```

```
SELECT ?p ?min ?value ?htime WHERE {

  ?range a med:AcceptableRange.
  ?range med:clinicalRangeMin ?min.
  ?range pd:hasParameter ?p.

  ?obs a mo:PhysiologicalObservation;
  ?obs ssn:observedProperty ?p;
  ?obs ssn:observationResultTime ?time;
  ?obs pd:atHumanTime ?htime;
  ?obs ssn:observedBy ?sensor.

  ?a1 ssn:hasValue ?a2 .
  ?a2 pd:readingValue ?value .

  ?sensor ssn:hasMeasurementCapability ?mc. ?mc a ssn:Accuracy;
  ms:capabilityValue ?accuracy.
  FILTER (?value < ?min)
  LET (?v2 := ?value * ?accuracy)
  FILTER ((?v2+?value) > ?min)
}
```

```
---query.aboveMinAcceptableMinusSensorAccuracy2
```

```
SELECT ?p ?min ?value ?htime WHERE {
  ?range a med:AcceptableRange; med:clinicalRangeMin ?min; pd:hasParameter ?p.
  ?obs a mo:PhysiologicalObservation; ssn:observedProperty ?p;
    ssn:observationResultTime ?time; pd:atHumanTime ?htime; ssn:observedBy ?sensor.
  ?obs ssn:observationResult/ssn:hasValue/pd:readingValue ?value.
  ?sensor ssn:hasMeasurementCapability ?mc. ?mc a ssn:Accuracy;
  ms:capabilityValue ?accuracy.
  FILTER (?value > ?min)
  LET (?v2 := ?value * ?accuracy)
  FILTER ((?value - ?v2) < ?min)
}
```

---query.belowMaxAcceptablePlusSensorAccuracy2

```
SELECT ?p ?max ?value ?htime WHERE {
  ?range a med:AcceptableRange; med:clinicalRangeMax ?max; pd:hasParameter ?p.
  ?obs a mo:PhysiologicalObservation; ssn:observedProperty ?p;
    ssn:observationResultTime ?time; pd:atHumanTime ?htime;
    ssn:observedBy ?sensor.
  ?obs ssn:observationResult/ssn:hasValue/pd:readingValue ?value.
  ?sensor ssn:hasMeasurementCapability ?mc. ?mc a ssn:Accuracy;
    ms:capabilityValue ?accuracy.
  FILTER (?value < ?max)
  LET (?v2 := ?value * ?accuracy)
  FILTER ((?v2+?value) > ?max)
}
```

---query.aboveMaxAcceptableMinusSensorAccuracy2

```
SELECT ?p ?max ?value ?htime WHERE {
  ?range a med:AcceptableRange; med:clinicalRangeMax ?max; pd:hasParameter ?p.
  ?obs a mo:PhysiologicalObservation; ssn:observedProperty ?p;
    ssn:observationResultTime ?time; pd:atHumanTime ?htime;
    ssn:observedBy ?sensor.
  ?obs ssn:observationResult/ssn:hasValue/pd:readingValue ?value.
  ?sensor ssn:hasMeasurementCapability ?mc. ?mc a ssn:Accuracy;
    ms:capabilityValue ?accuracy.
  FILTER (?value > ?max)
  LET (?v2 := ?value * ?accuracy)
  FILTER ((?value - ?v2) < ?max)
}
```

---query.aboveMaxAcceptableOkHypertension

```
SELECT ?obs ?p ?htime ?max ?value WHERE {
```

```
  ?range a med:AcceptableRange;
  med:clinicalRangeMax ?max;
  pd:hasParameter ?p.
```

```
  ?obs a mo:PhysiologicalObservation;
  ssn:observedProperty ?p;
  ssn:observationResultTime ?time;
  pd:atHumanTime ?htime.
  ?obs ssn:observationResult/ssn:hasValue/pd:readingValue ?value.
  FILTER (?value > ?max)
```

```
  med:Hypertension med:requiredSymptoms ?cs.
  ?cs med:clinicalFeatures ?cscf.
```

```
  ?cscf pd:hasParameter ?p.
  ?cscf med:clinicalRangeMax ?csrMax.
  ?cscf med:clinicalRangeMin ?csrMin.
```

```
  FILTER ((?value > ?csrMin)&&( ?value < ?csrMax))
```

```
}

---query.belowMinAcceptableOkHypotension

SELECT ?obs ?p ?htime ?min ?value WHERE {
  ?range a med:AcceptableRange; med:clinicalRangeMin ?min; pd:hasParameter ?p.
  ?obs a mo:PhysiologicalObservation;
  ssn:observedProperty ?p; ssn:observationResultTime ?time;
  pd:atHumanTime ?htime.
  ?obs ssn:observationResult/ssn:hasValue/pd:readingValue ?value.
  FILTER (?value < ?min)

  med:Hypotension med:requiredSymptoms ?cs.
  ?cs med:clinicalFeatures ?cscf.

  ?cscf pd:hasParameter ?p;
  ?cscf med:clinicalRangeMax ?csrMax;
  ?cscf med:clinicalRangeMin ?csrMin.

  FILTER ((?value > ?csrMin)&&( ?value < ?csrMax))
}
```

Appendix B

Appendix - Hadoop Cluster Configuration Files

B.1 HDFS Site

```
<configuration>
<property>
  <name>dfs.name.dir</name>
  <value>/home/hdfs/name</value>
</property>
<property>
  <name>dfs.data.dir</name>
  <value>/home/hdfs/data</value>
</property>
<property>
  <name>dfs.replication</name>
  <value>3</value>
</property>
</configuration>
```

B.2 Mapred Site

```
<configuration>
  <property>
    <name>mapred.job.tracker</name>
```

```
<value>192.168.53.1:9001</value>
</property>
<property>
<name>mapred.tasktracker.map.tasks.maximum</name>
<value>4</value>
</property>
<property>
<name>mapred.tasktracker.reduce.tasks.maximum</name>
<value>4</value>
</property>
<property>
<name>mapreduce.map.java.opts</name>
<value>-Xmx600m</value>
</property>
<property>
<name>mapreduce.reduce.java.opts</name>
<value>-Xmx600m</value>
</property>
<property>
<name>mapred.job.reuse.jvm.num.tasks</name>
<value>-1</value>
</property>
</configuration>
```

B.3 Core Site

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://192.168.53.1:9000/</value>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/home/tmp/hadoop-${user.name}</value>
    <description>A base for other temporary directories.</description>
  </property>
  <property>
    <name>io.file.buffer.size</name>
    <value>131072</value>
  </property>
</configuration>
```

Appendix C

Appendix - Approach One Source Code

C.1 Upload Algorithm

C.1.1 Map 1 - Compressor

```
package Mappers;

import java.io.IOException;
import java.util.Hashtable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import Utils.Parser;

public class Compressor extends Mapper<LongWritable, Text, Text, Text>
{
    Hashtable<String, String> replace = new Hashtable<String, String>();

    public void setup(Context context) throws IOException, InterruptedException
    {
        replace.put("<http://www.abdn.ac.uk/~csc316/ontologies/ICUNeuro.owl#" , "<med");
        replace.put("<http://www.abdn.ac.uk/~csc316/ontologies/PatientData#" , "<pd");
        replace.put("<http://www.abdn.ac.uk/~csc316/ontologies/MedicalObservations#" , "<mo");
        replace.put("<http://www.abdn.ac.uk/~csc316/ontologies/Annotations#" , "<ann");
        replace.put("<http://www.abdn.ac.uk/~csc316/ontologies/MedicalSensors#" , "<ms");
    }
}
```

```

        replace.put("<http://www.abdn.ac.uk/~csc316/ontologies/ObservationCollection#", "<oc");
        replace.put("<http://www.abdn.ac.uk/~csc316/ontologies/Specification#", "<spec");
        replace.put("<http://www.w3.org/2001/XMLSchema#", "<xsd");
        replace.put("<http://www.w3.org/2000/01/rdf-schema#", "<rdfs");
    }

    public void map(LongWritable key, Text value, Context context) throws IOException,
        InterruptedException
    {

        String [] Check = { "<http://www.w3.org/1999/02/22-rdf-syntax-ns" ,
            "<http://purl.oclc.org/NET/ssnx/ssn/" ,
            "<http://localhost:8085/data/" };

        String [] Replace = { "<a>",
            "<ssn",
            "</" };

        String Triple [] = null;
        try {
            Triple = Parser.parseTripleDynamic(value.toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
        for(int i = 0; i < 3; i++)
        {
            String temp[] = Triple[i].split("#");
            if (Triple[i].startsWith(Check[2]))
            {
                Triple[i] = Replace[2] + ":" + Triple[i].substring(Check[2].length(), Triple[i].length());
            }

            else if (Triple[i].startsWith(Check[0]))
            {
                Triple[i] = Replace[0];
            }
            else if(replace.containsKey(temp[0]+"#"))
            {
                Triple[i] = replace.get(temp[0]+"#") + ":" + temp[1];
            }
            else if (Triple[i].startsWith(Check[1]))

```

```

{
    Triple[i] = Replace[1] + ":" + Triple[i].substring(Check[1].length(), Triple[i].length()
    );
}
}
context.write(new Text(Triple[0] + " " + Triple[1] + " " + Triple[2]), new Text());
}
}

```

C.1.2 Map 2 - Create Single Line On Subject

```

package Mappers;

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class Data_Uploader_Pass1 extends Mapper<LongWritable, Text, Text, Text>
{
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException
    {
        String Triple[] = null;
        try {
            Triple = Utils.Parser.parseTripleDynamic(value.toString());
        } catch (Exception e) {
            e.printStackTrace();
        }

        word.set(Triple[0]);
        context.write(word, new Text(Triple[1] + " " + Triple[2]));

    }
}

```


C.1.3 Reduce 1 - Create Single Line On Subject

```
package Reducers;

import java.io.IOException;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.io.*;

public class Data_Uploader_Reduce1 extends Reducer<Text, Text, Text, Text>
{
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
        InterruptedException
    {
        String List = "";

        for (Text val : values)
        {
            String str = val.toString();
            List = List + str + " ";
        }

        context.write(new Text(key + " " + List), new Text());
    }
}
```

C.2 Query Algorithm

C.2.1 Map 1 - Pass 1

```
package Mappers;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.output.MultipleOutputs;
```

```

@SuppressWarnings("unused")
public class Pass_1 extends Mapper<LongWritable, Text, Text, Text>
{
    private static final Text obs = new Text("1"), a2 = new Text("2"), range = new Text("3"),
        mc = new Text("4"), cs = new Text("5");

    private MultipleOutputs<Text, Text> mos;

    public void setup(Context context){
        mos = new MultipleOutputs<Text, Text>(context);
    }

    public void map(LongWritable key, Text value, Context context) throws IOException,
        InterruptedException
    {
        Text joinKey = new Text();
        ArrayList<String> dataElementList = new ArrayList<String>();

        try {
            dataElementList = new ArrayList<String>(Utils.Parser.parseTripleDynamic(value.toString())
            );
        } catch (Exception e) {
            e.printStackTrace();
        }

        if (dataElementList.contains("<med:AcceptableRange>") &&
            dataElementList.contains("<med:clinicalRangeMax>") &&
            dataElementList.contains("<med:clinicalRangeMin>") &&
            dataElementList.contains("<pd:hasParameter>"))
        {

            String Max = dataElementList.get(dataElementList.indexOf("<med:clinicalRangeMax>")+1);
            String Min = dataElementList.get(dataElementList.indexOf("<med:clinicalRangeMin>")+1);
            String p = dataElementList.get(dataElementList.indexOf("<pd:hasParameter>")+1);

            mos.write("range", new Text(p + " " + Min + " " + Max + " " + dataElementList.get(0)),
                new Text());
        }

        else if (dataElementList.contains("<mo:PhysiologicalObservation>") &&
            dataElementList.contains("<ssn:observedProperty>") &&
            dataElementList.contains("<ssn:observationResultTime>") &&
            dataElementList.contains("<pd:atHumanTime>") &&

```

```

        dataElementList.contains("<ssn:observationResult>") &&
        dataElementList.contains("<ssn:observationResult>"))
    {
        String obs = dataElementList.get(0);
        String p = dataElementList.get(dataElementList.indexOf("<ssn:observedProperty>")+1);
        String hTime = dataElementList.get(dataElementList.indexOf("<pd:atHumanTime>")+1);
        String a1 = dataElementList.get(dataElementList.indexOf("<ssn:observationResult>")+1);
        String sensor = dataElementList.get(dataElementList.indexOf("<ssn:observedBy>")+1);

        mos.write("obs", new Text(obs + " " + p + " " + hTime + " " + a1 + " " + sensor), new
            Text());
    }

    else if(dataElementList.contains("<ssn:hasValue>"))
    {
        context.write(new Text(dataElementList.get(dataElementList.indexOf("<ssn:hasValue>")+1)),
            new Text(a2+"0"+dataElementList.get(0)));
    }

    else if(dataElementList.contains("<pd:readingValue>"))
    {
        context.write(new Text(dataElementList.get(0)), new Text(a2+"1"+dataElementList.get(
            dataElementList.indexOf("<pd:readingValue>")+1)));
    }

    else if(dataElementList.contains("<ssn:hasMeasurementCapability>"))
    {
        joinKey = new Text(dataElementList.get(dataElementList.indexOf("<ssn:
            hasMeasurementCapability>")+1));
        context.write(joinKey, new Text(mc+"0"+dataElementList.get(0)));
    }

    else if(dataElementList.contains("<ssn:Accuracy>") &&
        dataElementList.contains("<ms:capabilityValue>"))
    {
        joinKey = new Text(dataElementList.get(0));
        context.write(joinKey, new Text(mc+"1"+dataElementList.get(dataElementList.indexOf("<ms:
            capabilityValue>")+1)));
    }

    else if(dataElementList.get(0).equals("<med:Hypertension>") || dataElementList.get(0).
        equals("<med:Hypotension>"))
    {

```

```

    joinKey = new Text(dataElementList.get(dataElementList.indexOf("<med:requiredSymptoms>")+1));

    context.write(joinKey, new Text("5b"));
}

else if(dataElementList.contains("<med:clinicalFeatures>"))
{
    joinKey = new Text(dataElementList.get(0));

    context.write(joinKey, new Text(cs+"0"+dataElementList.get(dataElementList.indexOf("<med:clinicalFeatures>")+1)));
}

else if(dataElementList.contains("<med:clinicalRangeMax>") &&
    dataElementList.contains("<med:clinicalRangeMin>") &&
    dataElementList.contains("<pd:hasParameter>"))
{

    String Max = dataElementList.get(dataElementList.indexOf("<med:clinicalRangeMax>")+1);
    String Min = dataElementList.get(dataElementList.indexOf("<med:clinicalRangeMin>")+1);
    String p = dataElementList.get(dataElementList.indexOf("<pd:hasParameter>")+1);

    mos.write("cscf", new Text(p + " " + Max + " " + Min + " " + dataElementList.get(0)), new
        Text());
}
}

public void cleanup(Context context) throws IOException, InterruptedException
{
    mos.close();
}
}

```

C.2.2 Reduce 1 - Pass 1

```

package Reducers;

import java.io.IOException;
import java.util.HashSet;
import java.util.Set;

import org.apache.hadoop.io.NullWritable;

```

```

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.Reducer.Context;
import org.apache.hadoop.mapreduce.lib.outputMultipleOutputs;

@SuppressWarnings("unused")
public class Reduce_1 extends Reducer<Text, Text, Text, Text>
{

    private MultipleOutputs<Text, Text> mos;

    public void setup(Context context)
    {
        mos = new MultipleOutputs<Text, Text>(context);
    }

    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
        InterruptedException
    {
        Utils.DataObject a2 = new Utils.DataObject();
        Utils.DataObject mc = new Utils.DataObject();
        Utils.DataObject cs = new Utils.DataObject();

        for (Text val : values)
        {
            String str = val.toString();

            switch(str.charAt(0))
            {
                //Check for ?a2 -----
                case '2':
                    a2.addElementString(str.substring(1));
                    break;

                //Begin checking for ?mc -----
                case '4':
                    mc.addElementString(str.substring(1));

                    break;

                //Begin cheking for ?cs -----
                case '5':
                    str = str.substring(1);

```

```

        if (str.equals("b"))
        {
            cs.addElementBool(true);
        }
        else
        {
            cs.addElementString(str);
        }
        break;
    }
}

if (!a2.isStringEmpty() && a2.isStringLength(2))
{
    String[] sorted = a2.sortString();

    mos.write("obs", new Text(sorted[0] + Utils.Constants.W_SPACE +
        sorted[1]), new Text());
}
else if (!mc.isStringEmpty() && mc.isStringLength(2))
{
    String[] sorted = mc.sortString();

    mos.write("mc", new Text(sorted[0] + Utils.Constants.W_SPACE +
        sorted[1]), new Text());
}

// Output ?cs with the prefix <cs
>
else if (cs.isBoolTrue() && !cs.isStringEmpty())
{
    String[] sorted = cs.sortString();

    mos.write("cs", new Text(sorted[0]), new Text());
}

public void cleanup(Context context) throws IOException, InterruptedException
{
    mos.close();
}
}

```

C.2.3 Map 2 - Pass 2

```
package Mappers;
import java.io.IOException;
import java.util.ArrayList;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class Pass_2 extends Mapper<LongWritable, Text, Text, Text>
{
    private static final Text one = new Text("1"), two = new Text("2");

    public void map(LongWritable key, Text value, Context context) throws IOException,
        InterruptedException
    {
        Text joinKey = new Text();
        ArrayList<String> dataElementList = new ArrayList<String>();

        try {
            dataElementList = new ArrayList<String>(Utils.Parser.parseTripleDynamic(value.toString()));
        } catch (Exception e) {
            e.printStackTrace();
        }

        if (dataElementList.size() == 2)
        {
            joinKey.set(dataElementList.get(0));
            context.write(joinKey, new Text(one + dataElementList.get(1)));
        }
        else if (dataElementList.size() == 5)
        {
            joinKey.set(dataElementList.get(3));
            context.write(joinKey, new Text(two + dataElementList.get(0) + Utils.Constants.WSPACE +
                dataElementList.get(1) + Utils.Constants.WSPACE +
                dataElementList.get(2) + Utils.Constants.WSPACE +
                dataElementList.get(4)));
        }
    }
}
```

C.2.4 Reduce 2 - Pass 2

```

package Reducers;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;
import java.util.Hashtable;
import java.util.Set;
import java.util.Vector;

import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.Reducer.Context;
import org.apache.hadoop.mapreduce.lib.output.MultipleOutputs;

@SuppressWarnings("unused")
public class Reduce_2 extends Reducer<Text, Text, Text, Text>
{
    Path[] file;
    static enum Error { BELOW.MIN, ABOVE.MAX, BMIN.P.SENSOR, BMIN.M.SENSOR, BMAX.P.SENSOR,
        BMAX.M.SENSOR, HYPOTENSION, HYPERTENSION }

    Set<String> cscfList = new HashSet<String>();
    Set<String> csList = new HashSet<String>();

    Hashtable<String, Vector<Float>> rangeHashMap = new Hashtable<String, Vector<Float>>();
    Hashtable<String, Vector<Float>> cscfFinalHashMap = new Hashtable<String, Vector<Float>>();
    Hashtable<String, Float> mcHashMap = new Hashtable<String, Float>();

    public MultipleOutputs<Text, Text> mos;

    //Parse values from the distro cache
    public void setup(Context context) throws IOException, InterruptedException
    {

        // Create the output for use in reduce
        mos = new MultipleOutputs<Text, Text>(context);

        final String CSCF.FILENAME = "cscf-m";
        final String CS.FILENAME = "cs-r-00000";
        final String RANGE.FILENAME = "range-m-";
        final String MC.FILENAME = "mc-r-00000";
    }
}

```



```

Path[] files = DistributedCache.getLocalCacheFiles(context.getConfiguration());

for (Path p : files)
{
    if (p.getName().contains(CSCF_FILENAME))
    {
        BufferedReader reader = new BufferedReader(new FileReader(p.toString()));
        String line = reader.readLine();
        while(line != null)
        {
            cscfList.add(line);
            line = reader.readLine();
        }
        reader.close();
    }
    else if (p.getName().contains(CS_FILENAME))
    {
        BufferedReader reader = new BufferedReader(new FileReader(p.toString()));
        String line = reader.readLine();
        while(line != null)
        {
            csList.add(line);
            line = reader.readLine();
        }
        reader.close();
    }
    else if (p.getName().contains(RANGE_FILENAME))
    {
        BufferedReader reader = new BufferedReader(new FileReader(p.toString()));
        String line = reader.readLine();
        while(line != null)
        {
            // <p> <min> <max>
            String [] temp1 = line.split(" ");
            Vector<Float> v = new Vector<Float>();
            v.add(Float.parseFloat(temp1[1].substring(1,temp1[1].indexOf('')+1)));
            v.add(Float.parseFloat(temp1[2].substring(1,temp1[2].indexOf('')+1)));
            rangeHashMap.put(temp1[0], v);

            line = reader.readLine();
        }
        reader.close();
    }
}

```

```

    }
    else if (p.getName().contains(MC.FILENAME))
    {
        BufferedReader reader = new BufferedReader(new FileReader(p.toString()));
        String line = reader.readLine();
        while(line != null)
        {
            String [] temp1 = line.split(" ");
            mcHashMap.put(temp1[0], Float.parseFloat(temp1[1].substring(1,temp1[1].substring(1).
            indexOf('')+1)));

            line = reader.readLine();
        }
        reader.close();
    }
}

Vector<Float> v = new Vector<Float>();

//Join <cs> to <cscf> to create the final cscf to be used in the main function for
    joining.
for( String cscf : cscfList){
    for( String cs : csList)
    {
        if(cscf.contains(cs.trim()))
        {
            String [] temp1 = cscf.split(" ");
            v.add(Float.parseFloat(temp1[1].substring(1,temp1[1].substring(1).indexOf('')+1)));
            v.add(Float.parseFloat(temp1[2].substring(1,temp1[2].substring(1).indexOf('')+1)));

            cscfFinalHashMap.put(temp1[0], v);
        }
    }
}
cscfList.clear();
csList.clear();
}

public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
    InterruptedException
{
    Set<String> obsList = new HashSet<String>();
    Set<String> a2List = new HashSet<String>();

```

```

String temp1[] = null, obs = null, p = null, htime = null, sensor = null;
float value = 0, min = 0, max = 0, accuracy = 0, v2 = 0, csrMax = 0, csrMin = 0;

// Parse values
for (Text val : values)
{
    String str = val.toString();
    switch( str.charAt(0))
    {
        case '1': a2List.add(str.substring(1)); break;
        case '2': obsList.add(str.substring(1)); break;
    }
}
for (String obsE : obsList) {
    for (String valueS : a2List)
    {
        temp1 = obsE.split(" ");

        // Extract variables from the ?obs string
        obs = temp1[0];
        p = temp1[1];
        htime = temp1[2] + Utils.Constants.W.SPACE + temp1[3];
        sensor = temp1[4];

        value = Float.parseFloat(valueS.substring(1,valueS.substring(1).indexOf('')+1));

        //Parse the rangeList and split the elements
        if(rangeHashMap.containsKey(p))
        {
            Vector<Float> temp = rangeHashMap.get(p);
            min = temp.elementAt(0);
            max = temp.elementAt(1);
        }
        else
        {
            continue;
        }

        // Perform Min and Max Queries


---


        if (value < min)
        {

```

```

//Output the final result in following format <?obs ?htime ?p ?value ?min>
mos.write("BelowMin", new Text(obs + Utils.Constants.W_SPACE +
    p + Utils.Constants.W_SPACE +
    htime + Utils.Constants.W_SPACE +
    Float.toString(min) + Utils.Constants.W_SPACE +
    valueS), new Text());

context.getCounter(Error.BELLOW_MIN).increment(1);
}
else if (value > max)
{

//Output the final result in following format <?obs ?htime ?p ?value ?max>
mos.write("AboveMax", new Text(obs + Utils.Constants.W_SPACE +
    p + Utils.Constants.W_SPACE +
    htime + Utils.Constants.W_SPACE +
    Float.toString(max) + Utils.Constants.W_SPACE +
    valueS), new Text());

context.getCounter(Error.ABOVE_MAX).increment(1);
}

// Begin accuracy checks


---


if (mcHashMap.containsKey(sensor))
{
    accuracy = mcHashMap.get(sensor);
    v2 = value * accuracy;
}
else
{
    continue;
}

// Below Min Plus Sensor Accuracy
if ((value < min) && ((v2 + value) > min))
{
    //Output the final result in following format <?obs ?htime ?p ?value ?min>
mos.write("BelowMinPlusSensorAccuracy", new Text( obs + Utils.Constants.W_SPACE +
    p + Utils.Constants.W_SPACE +
    htime + Utils.Constants.W_SPACE +
    Float.toString(min) + Utils.Constants.W_SPACE +
    valueS), new Text());

```

```

    context.getCounter(Error.BMIN.P.SENSOR).increment(1);
}

//Below Min Minus Sensor Accuracy
else if ((value > min) && ((value - v2) < min))
{
    //Output the final result in following format <?obs ?htime ?p ?value ?min>
    mos.write("BelowMinMinusSensorAccuracy" ,new Text(obs + Utils.Constants.W_SPACE +
        p + Utils.Constants.W_SPACE +
        htime + Utils.Constants.W_SPACE +
        Float.toString(min) + Utils.Constants.W_SPACE +
        valueS), new Text());

    context.getCounter(Error.BMIN.M.SENSOR).increment(1);
}

// Below Max Plus Sensor Accuracy
if ((value < max) && ((v2 + value) > max))
{
    //Output the final result in following format <?obs ?htime ?p ?value ?min>
    mos.write("BelowMaxPlusSensorAccuracy" ,new Text(obs + Utils.Constants.W_SPACE +
        p + Utils.Constants.W_SPACE +
        htime + Utils.Constants.W_SPACE +
        Float.toString(max) + Utils.Constants.W_SPACE +
        valueS), new Text());

    context.getCounter(Error.BMAX.P.SENSOR).increment(1);
}

// Above Max Minus Sensor Accuracy
else if ((value > max) && ((value - v2) < max))
{
    //Output the final result in following format <?obs ?htime ?p ?value ?min>
    mos.write("AboveMaxPlusSensorAccuracy" ,new Text(obs + Utils.Constants.W_SPACE +
        p + Utils.Constants.W_SPACE +
        htime + Utils.Constants.W_SPACE +
        Float.toString(max) + Utils.Constants.W_SPACE +
        valueS), new Text());

    context.getCounter(Error.BMAX.M.SENSOR).increment(1);
}

// Begin checks based on Hypertention and Hypotension

```

```

    if (cscfFinalHashMap.containsKey(p))
    {
        Vector<Float> temp = cscfFinalHashMap.get(p);

        for(int i = 0; i < 4; i=i+2)
        {
            csrMax = temp.elementAt(i);
            csrMin = temp.elementAt(i+1);

            if ((value < min) && (value > csrMin) && (value < csrMax))
            {
                //Output the final result in following format <Value Below Minimum: Hypertension>
                mos.write("ValueBelowMinimumHypotension" ,new Text(obs + Utils.Constants.W.SPACE +
                    p + Utils.Constants.W.SPACE +
                    htime + Utils.Constants.W.SPACE +
                    Float.toString(min) + Utils.Constants.W.SPACE +
                    valueS), new Text());

                context.getCounter(Error.HYPOTENSION).increment(1);
            }

            if ((value > max) && (value > csrMin) && (value < csrMax))
            {
                //Output the final result in following format <?obs ?htime ?p ?value ?min>
                mos.write("ValueAboveMaximumHypertension" ,new Text(obs + Utils.Constants.W.SPACE +
                    p + Utils.Constants.W.SPACE +
                    htime + Utils.Constants.W.SPACE +
                    Float.toString(max) + Utils.Constants.W.SPACE +
                    valueS), new Text());

                context.getCounter(Error.HYPERTENSION).increment(1);
            }
        }
    }
    else
    {
        continue;
    }
}

}

public void cleanup(Context context) throws IOException, InterruptedException
{

```

```
mos.close();  
}  
}
```

Appendix D

Appendix - Approach Two Source Code

D.1 Query Algorithm

D.1.1 Map 1 - Pass 1

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

@SuppressWarnings("unused")
public class Pass_1 extends Mapper<LongWritable, Text, Text, Text>
{
    private static final Text obs = new Text("1"), a2 = new Text("2"), range = new Text("3"),
        mc = new Text("4"), cs = new Text("5");

    public void map(LongWritable key, Text value, Context context) throws IOException,
        InterruptedException
    {
        Text word = new Text();
        String Triple[] = null;

        try {
            Triple = Utils.Parser.parseTripleDynamic(value.toString());
```



```

} catch (Exception e) {
    e.printStackTrace();
}

// The output format is (item class – item order – item)
if (Triple[1].equals(Utils.Constants.RDF.TYPE))
{
    if (Triple[2].equals(Utils.Constants.MED_URI.AcceptableRange))
    {
        word.set(Triple[0]);
        context.write(word, new Text(range + "b"));
    }
    else if (Triple[2].equals(Utils.Constants.SSN_URI.Accuracy))
    {
        word.set(Triple[0]);
        context.write(word, new Text(mc + "b"));
    }
    else if (Triple[2].equals(Utils.Constants.MO_URI.PhysiologicalObservation))
    {
        word.set(Triple[0]);
        context.write(word, new Text(obs + "b"));
    }
}
else if (Triple[1].equals(Utils.Constants.MED_URI.requiredSymptoms))
{
    if (Triple[0].equals(Utils.Constants.MED_URI.Hypertension) || Triple[0].equals(Utils.
        Constants.MED_URI.Hypotension))
    {
        word.set(Triple[2]);
        context.write(word, new Text(cs + "b"));
    }
}
else if (Triple[1].startsWith(Utils.Constants.SSN.ONTO.PREFIX))
{
    if (Triple[1].endsWith(Utils.Constants.SSN_URI.observedProperty))
    {
        word.set(Triple[0]);
        context.write(word, new Text(obs + "2" + Triple[2]));
    }
    else if (Triple[1].endsWith(Utils.Constants.SSN_URI.observationResultTime))
    {
        word.set(Triple[0]);
        context.write(word, new Text(obs + "b"));
    }
}

```

```

else if (Triple [1].endsWith( Utils . Constants . SSN_URI_observationResult))
{
    word.set( Triple [0]);
    context.write( word, new Text( obs + "0" + Triple [2]));
}
else if (Triple [1].endsWith( Utils . Constants . SSN_URI_observedBy))
{
    word.set( Triple [0]);
    context.write( word, new Text( obs + "3" + Triple [2]));
}
else if (Triple [1].endsWith( Utils . Constants . SSN_URI_hasValue))
{
    word.set( Triple [2]);
    context.write( word, new Text( a2 + "0" + Triple [0]));
}
else if (Triple [1].endsWith( Utils . Constants . SSN_URI_hasMeasurementCapability))
{
    word.set( Triple [2]);
    context.write( word, new Text( mc + "0" + Triple [0]));
}
}
else if (Triple [1].startsWith( Utils . Constants . MED_ONTO_PREFIX))
{
    if (Triple [1].endsWith( Utils . Constants . MED_URI_clinicalRangeMin))
    {
        word.set( Triple [0]);
        context.write( word, new Text( range + "1" + Triple [2]));
    }
    else if (Triple [1].endsWith( Utils . Constants . MED_URI_clinicalRangeMax))
    {
        word.set( Triple [0]);
        context.write( word, new Text( range + "2" + Triple [2]));
    }
    else if (Triple [1].endsWith( Utils . Constants . MED_URI_clinicalFeatures))
    {
        word.set( Triple [0]);
        context.write( word, new Text( cs + "0" + Triple [2]));
    }
}
else if (Triple [1].startsWith( Utils . Constants . PD_ONTO_PREFIX))
{
    if (Triple [1].endsWith( Utils . Constants . PD_URI_atHumanTime))
    {

```

```

    word.set(Triple[0]);
    context.write(word, new Text(obs + "1" + Triple[2]));
}
else if (Triple[1].endsWith(Utils.Constants.PD_URI_readingValue))
{
    word.set(Triple[0]);
    context.write(word, new Text(a2 + "1" + Triple[2]));
}
    else if (Triple[1].endsWith(Utils.Constants.PD_URI_hasParameter))
{
    word.set(Triple[0]);
    context.write(word, new Text(range + "0" + Triple[2]));
}
}
else if (Triple[1].equals(Utils.Constants.SEN_URI_PhysiologicalObservation))
{
    word.set(Triple[0]);
    context.write(word, new Text(mc + "1" + Triple[2]));
}
}
}

```

D.1.2 Reduce 1 - Pass 1

```

import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.Reducer.Context;
import org.apache.hadoop.mapreduce.lib.outputMultipleOutputs;

@SuppressWarnings("unused")
public class Reduce_1 extends Reducer<Text, Text, Text, Text>
{
    private MultipleOutputs<Text, Text> mos;
    public void setup(Context context){
        mos = new MultipleOutputs<Text, Text>(context);
    }
}

```

```

public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
    InterruptedException
{

    Utils.DataObject range = new Utils.DataObject();
    Utils.DataObject cs = new Utils.DataObject();
    Utils.DataObject a2 = new Utils.DataObject();
    Utils.DataObject mc = new Utils.DataObject();
    Utils.DataObject obs = new Utils.DataObject();

    for (Text val : values)
    {
        String str = val.toString();

        switch(str.charAt(0))
        {
            // Check for ?obs -----
            case '1':
                str = str.substring(1);
                if (str.equals("b"))
                {
                    obs.addElementBool(true);
                }
                else
                {
                    obs.addElementString(str);
                }
                break;

            //Check for ?a2 -----
            case '2':
                a2.addElementString(str.substring(1));
                break;

            //Begin checking for ?range -----
            case '3':
                str = str.substring(1);

                if (str.equals("b"))
                {
                    range.addElementBool(true);
                }
                else
                {

```

```

        range.addElementString(str);
    }
    break;

//Begin checking for ?mc
case '4':
    str = str.substring(1);

    if(str.equals("b"))
    {
        mc.addElementBool(true);
    }
    else
    {
        mc.addElementString(str);
    }
    break;

//Begin cheking for ?cs
case '5':
    str = str.substring(1);

    if(str.equals("b"))
    {
        cs.addElementBool(true);
    }
    else
    {
        cs.addElementString(str);
    }
    break;
}
}

// Output ?obs with the prefix <obs>
if(obs.isBoolTrue() && !obs.isStringEmpty())
{
    String[] sorted = obs.sortString();

    mos.write("obs", new Text(key + Utils.Constants.W_SPACE +
        sorted[0] + Utils.Constants.W_SPACE +
        sorted[1] + Utils.Constants.W_SPACE +
        sorted[2] + Utils.Constants.W_SPACE +
        sorted[3]), new Text());
}

```

```

}

// Output ?a2 with the prefix <a2>


---


else if (!a2.isStringEmpty() && a2.isStringLength(2))
{
    String[] sorted = a2.sortString();

    mos.write("obs", new Text(sorted[0] + Utils.Constants.W_SPACE +
        sorted[1]), new Text());
}

// Output ?range with the prefix <range>


---


else if (range.
    isBoolTrue() && !range.isStringEmpty())
{
    String[] sorted = range.sortString();

    mos.write("range", new Text(sorted[0] + Utils.Constants.W_SPACE +
        sorted[1] + Utils.Constants.W_SPACE +
        sorted[2]), new Text());
}

// Output ?mc with the prefix <mc>


---


else if (mc.isBoolTrue() && !mc.isStringEmpty())
{
    String[] sorted = mc.sortString();

    mos.write("mc", new Text(sorted[0] + Utils.Constants.W_SPACE +
        sorted[1]), new Text());
}

// Output ?cs with the prefix <cs>


---


else if (cs.isBoolTrue() && !cs.isStringEmpty())
{
    String[] sorted = cs.sortString();

    mos.write("cs", new Text(sorted[0]), new Text());
}

```

```
}

// Output ?cscf with the prefix <cscf
>
else if (!range.isStringEmpty() && range.isStringLength(3))
{
    String[] sorted = range.sortString();

    mos.write("cscf", new Text (key + Utils.Constants.W_SPACE +
        sorted[0] + Utils.Constants.W_SPACE +
        sorted[2] + Utils.Constants.W_SPACE +
        sorted[1]), new Text());
}
}

public void cleanup(Context context) throws IOException, InterruptedException
{
    mos.close();
}
}
```

D.1.3 Map 2 - Pass 2

The source code for this is the same as for approach one's and can be found in Appendix C.

D.1.4 Reduce 2 - Pass 2

The source code for this is the same as for approach one's and can be found in Appendix C.